MIT Device Simulation WebLab: An Online Simulator for Microelectronic Devices

by

Adrian Solis

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2004 [June 2005]

© Adrian Solis, MMIV. All rights reserved.

The author hereby grants to MIT permission to reproduce and distribute publicly paper and electronic copies of this thesis document in whole or in part.

Author .....
Department of Electrical Engineering and Computer Science
September 3, 2004

Certified by .....
Jesús A. del Alamo
Professor of Electrical Engineering
Thesis Supervisor

Accepted by
Arthur C. Smith
Chairman, Department Committee on Graduate Students

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
JUL 18 2005
LIBRARIES

BARKER

MIT Device Simulation WebLab: An Online Simulator for Microelectronic Devices

by

Adrian Solis

Submitted to the Department of Electrical Engineering and Computer Science
on September 3, 2004, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

In the field of microelectronics, a device simulator is an important engineering tool with tremendous educational value. With a device simulator, a student can examine the characteristics of a microelectronic device described by a particular model. This makes it easier to develop an intuition for the general behavior of that device and examine the impact of particular device parameters on device characteristics.

In this thesis, we designed and implemented the MIT Device Simulation WebLab ("WeblabSim"), an online simulator for exploring the behavior of microelectronic devices. WeblabSim makes a device simulator readily available to users on the web anywhere, and at any time. Through a Java applet interface, a user connected to the internet specifies and submits a simulation to the system. A program performs the simulation on a computer that can be located anywhere else on the internet. The results are then sent back to the user's applet for graphing and further analysis.

The WeblabSim system uses a three-tier design based on the iLab Batched Experiment Architecture. It consists of a client applet that lets users configure simulations, a laboratory server that runs them, and a generic service broker that mediates between the two through SOAP-based web services. We have implemented a graphical client applet, based on the client used by the MIT Microelectronics WebLab. Our laboratory server has a distributed, modular design consisting of a data store, several worker servers that run simulations, and a master server that acts as a coordinator. On this system, we have successfully deployed WinSpice, a circuit simulator based on Berkeley Spice3F4.

Our initial experiences with WeblabSim indicate that it is feature-complete, reliable and efficient. We are satisfied that it is ready for beta deployment in a classroom setting, which we hope to do in Fall 2004.

Thesis Supervisor: Jesús A. del Alamo
Title: Professor of Electrical Engineering

Acknowledgments

My thanks go to, first and foremost, my advisor Professor Jesús del Alamo. His abundant and insightful guidance, his sincere efforts to understand the details of my work, and his generous praise made this project a joyous task. He shepherded the MIT Microelectronics WebLab from a small on-campus project in 1998 to a sophisticated online laboratory used by many students all over the world today. Thus, I am confident that under his leadership, my project, the MIT Device Simulation WebLab, will reach its full potential.

Along these lines, I also thank the past and present members of the WebLab Group. In particular, my colleagues James Hardison and David Zych were especially helpful. Building on their work made my own so much easier. I feel that I now understand better what Newton meant when he said, "If I have seen further it is by standing on the shoulders of Giants."

I thank my parents, Teodoro and Naomi Solis, who have supported me every step of the way. From my childhood, they have worked hard to give me the best education possible. I am rather amazed that they had the courage to let me attend the Massachusetts Institute of Technology, which is literally halfway around the world from where we lived. Very few parents, especially in the Philippines, love their children so much that they let them grow in their own way, in their own time, and in their own place. For that I am truly grateful.

I also thank my wonderful friends—Hubert Pham, Lee Lin, Kathryn Chen, and Xian Ke, most of all—who have been there for me from the day that we met. With them, I experienced more than enough happiness to keep my spirit alive, and just enough pain that the smallest joys in my life appeared much bigger.

Finally, I thank Bradley Jellerichs, who managed to turn what would otherwise have been two weeks of dreary thesis-writing into a truly magical time. I thought getting Professor del Alamo's signature on this thesis would be the highlight of my final days at MIT; it turns out greater things were destined to happen. Brad, thank you very much.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

A simulator is an important engineering tool. Engineers use simulators to design everything from bridges to electrical circuits. Simulators are especially useful in microelectronics, where development cycles are long and expensive. After a certain point in the development process, mistakes become increasingly costly to fix. Thus, it is essential for circuit and device engineers to verify the correctness of their designs before they are fabricated.

In addition to being standard fare in the every day work of engineers, simulators have tremendous educational value. With a device simulator, a student can examine how the device described by a particular model behaves when presented with various inputs. Although this exploration can be done with a real device, the appropriate equipment is often prohibitively expensive. Whereas device simulators are available at reasonable prices – in fact, some are even freely available on the web [1].

Furthermore, working with a device simulator allows students to vary conditions which are hard to control in real devices. For instance, he may want to know the effect of the substrate doping level (N_{SUB}) on the threshold voltage (V_T) of an n-type MOSFET. To do this experiment using real devices, he has to identify N_{SUB} for a given transistor and obtain a series of transistors spanning a range of doping levels: a daunting task. A similar simulation is easily performed using SPICE [2].

Most programs can even be configured to graph the results. Visually examining how changes in a model parameter affect device characteristics makes it easier to develop an intuition into the physical effects that the parameter captures.

A device simulator lets students explore device behavior in regimes that would otherwise be infeasible or unsafe to examine. A classic example is the response of a transistor to voltage conditions that lead to destructive breakdown. Unless the experimental setup is configured correctly, sending a device into destructive breakdown carries the risk of device damage. Moreover, repeatedly sending a device outside its normal mode of operation can degrade its characteristics. The resulting drift makes it harder to compare results from two experiments performed at different times.

Finally, integrating simulations and experiments is a powerful way to learn about a device. Students can compare measurements from real devices with predictions derived from the models they learn in class. By doing this, they gain a better understanding of the conditions under which these idealized models are applicable, and where the models fail.

1.2 The MIT Device Simulation WebLab

This thesis describes the design and development of the MIT Device Simulation WebLab ("WeblabSim"), an online simulator for exploring the behavior of devices. Using a Java applet interface, a user connected to the web can specify and run simulations of microelectronic devices. To define a simulation, he chooses a device model, and then configures the behavior of the model's parameters and terminals. When submitted, a device simulator performs the simulation on computers located at MIT, and the results are then sent back to the user's applet for graphing and further analysis.

WeblabSim exposes device simulators through a "virtual laboratory" interface. The system presents device models to the user as if they were actual devices, albeit with variable parameters that control their behavior. From the user's point of view, the terminals of the virtual device are attached to the ports of an analyzer, which he can then program as he wishes. When he is satisfied with his setup, he submits

his experiment. He then receives the results of his experiment after it is seemingly performed at a remote location. The translation between the laboratory and simulator paradigms is seamless.

1.3 Related Work

1.3.1 Local Simulators

A stand-alone simulator can be considered to be the simplest virtual laboratory. In the field of microelectronics, there are various circuit simulators available. High-end programs like HSPICE™[3] and PSpice®[4] offer high performance, extensive support for a wide variety of device models, and advanced analyses (e.g., sensitivity analysis, Monte Carlo analysis, smoke analysis). Simpler programs provide a more limited set of features, but for a lower price. WinSpice [1] is a circuit simulator based on Berkeley Spice3F4 [2] that is free for most uses. A license key, which activates more advanced features like parameterized sub-circuits and PSpice libraries, is available from the author for £45.[1]

Some circuit simulators [5, 6] are packaged with an integrated environment that lets users create, simulate and export circuit designs. These programs have a rich interface that employs a circuit building metaphor: users compose circuits by picking components and connecting them onscreen. Students can build circuits and try them out, just as they would in a real laboratory, but without expense or danger. Similar software exists for other fields of engineering; examples include Working Model™[7] for mechanics and CyclePad [8] for engineering thermodynamics.

1.3.2 Online Simulators

Several companies and individuals have developed circuit simulators that they have put on the web for online use. National Semiconductor runs the WEBENCH Electrical Simulator [9], an online electrical simulation tool for National components.

[1]The price was obtained from http://www.winspice.com on August 16, 2004.

WEBENCH utilizes a standard SPICE simulation engine that is integrated with a graphical representation of the circuit schematic. The program displays the results in a waveform viewer, which allows multiple waveforms to be viewed together. This feature guides the user in optimizing his circuit by highlighting the differences between designs. Intersil has a similar online application called iSim [10]. Both programs restrict the user to running an AC analysis a set of predefined circuit layouts that use the company's technology.

DigSim [11] is a digital logic simulator developed by Iwan van Rienen. DigSim is a Java applet that allows users to create digital schematics in a Computer Aided Drawing (CAD) environment. The schematic can then be simulated to show how the circuit would perform if actually constructed. Mr. van Rienen has since stopped development on DigSim. However, a version of the program is still being maintained at the University of California in Berkeley [12].

1.3.3 Virtual Experiments on the World Wide Web

The Virtual Laboratory [13] at Johns Hopkins University is a collection of web-based science and engineering experiments developed for beginning students. The exercises were designed to expose high school seniors and college freshmen to experimentation, problem solving, data gathering, and scientific interpretation. Ordinarily, students were unable to practice these skills until they worked in a design laboratory in their junior or senior year. The activities cover a wide range of engineering topics, including logic circuits, heat transfer and sound propagation.

The Oxford Virtual Reality Group is hosting Virtual Chemistry [14], a three-dimensional simulated laboratory for the teaching of chemistry. The site interface was modelled using virtual reality techniques. The user moves around the laboratory in a web browser window and takes part in experiments distributed around this virtual environment. The experiments feature videos and animations of the procedure being carried out, three-dimensional simulations of chemical processes and interactive sections that check the student's understanding of the material.

1.3.4 MIT Microelectronics WebLab

The MIT Microelectronics WebLab ("WebLab") [15, 16, 17, 18] is an online remote laboratory for characterizing microelectronic devices. Through a Java-enabled web browser, users all over the world can run experiments on real transistors, diodes, and other devices by means of a semiconductor parameter analyzer located at MIT. WebLab has been very successful. Since its inception in 1998, over 1,900 students have used WebLab as part of graded assignments in classes at MIT, the National University of Singapore, and the Chalmers University of Technology in Sweden [19].

As an online remote laboratory in the microelectronics domain, WebLab has heavily influenced the WeblabSim user interface. The latest version of WebLab has a graphical representation of the experiment setup that users find to be intuitive [20, 21]. In fact, the features and user interface of WeblabSim were carefully designed to build on those of WebLab, so as to leverage WebLab's usability and familiarity.

Our ultimate goal is to integrate WebLab and WeblabSim into a single online laboratory. A student can then perform both measurements and simulations, and compare their results. We believe that the conjunction of an online remote laboratory and an online simulator will be a valuable resource in microelectronics education.

WeblabSim's design and implementation has been so heavily influenced both by WebLab itself, and its underlying system blueprint, the iLab Batched Experiment Architecture (BEA) [22]. Indeed, as we will discuss in Chapter 2, WeblabSim has adopted the iLab BEA in its own design. Because of this, we have adopted the laboratory notations of WebLab and the iLab BEA for the equivalent concepts in WeblabSim. For example, we often talk of WeblabSim as an "online laboratory", we call the system configuration the "laboratory configuration", and we refer to users' simulations as "experiments".

1.4 Thesis Outline

This thesis is organized as follows. Chapter two describes the problem being solved, and the solution. It presents a description of the underlying problem, derives reason-

able design goals, and then gives an overview of the chosen system architecture.

The succeeding chapters focus on the individual parts of WeblabSim. Chapter three describes the client applet. It presents the user's view of the system, and discuses its design and implementation. Chapter four focuses on the design of the laboratory server. It enumerates the components that comprise the laboratory server, and explains their interactions with each other and with the rest of the system.

Finally, we conclude by summarizing our main accomplishments and discussing our vision for future work on the MIT Microelectronics Device Simulation WebLab.

Chapter 2

The MIT Device Simulation WebLab

This chapter describes the problem addressed by the MIT Device Simulation WebLab, and its solution. It begins by analyzing the problem domain, resulting in a series of conceptual models on which we based the design of the system and the user interface. The following sections establish design goals based on the values of extensibility, reliability and efficiency, and then introduce a design blueprint that allows WeblabSim to meet these targets. It concludes with a high-level description of the system, which will be elaborated further in succeeding chapters.

2.1 Problem Analysis

The problem WeblabSim addresses, stated in general terms, is the design and implementation of an online device simulator. Solving this problem successfully requires understanding it. In this section, we carefully analyze the problem domain by identifying key abstractions and defining the relationships between them. This analysis will clarify the conceptual model behind the user interface presented above, and the design as it is described in the following chapters.

2.1.1 Device and Device Type

A *device* is a circuit component. Devices have at least two *terminals*, by which they are connected to the rest of a circuit. The voltage at a terminal and the current flowing into a terminal are numerical quantities that can be measured. Devices can have *observables*, which are named numerical quantities associated with the present state of the device. An example of an observable would be the dynamic resistance of a pn diode. Together, the values of the voltages and currents at the terminals of a device and the values of all the its observables comprise the device's state. We refer to these as the *components* of the device state.

A *device type* is a class of devices with similar behaviors. For instance, the class of devices with two terminals and that to the first order obey Ohm's law may be grouped into a device type called "resistor". All devices of a given type necessarily have the same number of terminals, and the same set of observables.

Figure 2-1 shows the relationships between devices, device types and terminals in an object model. The constraints on the relationships are expressed using the Alloy object modelling notation [23]. A box denotes a set of objects. A closed arrow from box A to box B denotes a subset relationship, that is, A is a subset of B. Open arrows describe relationships between sets, and are labelled with the name of the relationship. Either end of a relationship may be qualified by a hash mark, which indicates immutability, or by a multiplicity marker (* = zero or more, ? = zero or one, ! = exactly one, + = one or more).

2.1.2 Device Model

A *device model* is a set of relations between the components of a device state.[1] The relations are typically expressed as equations involving the state components, constants, and adjustable *parameters*. The equations that comprise a model may have been derived from first principles, or obtained empirically from experimental results.

[1] Although we present the concepts abstractly, the notation introduced in this and the preceding section suggest a concrete representation. The device state can be represented as a vector, with the terminal currents, terminal voltages and other observables as the vector's components. A device model is then a multi-valued function acting on the state vector.

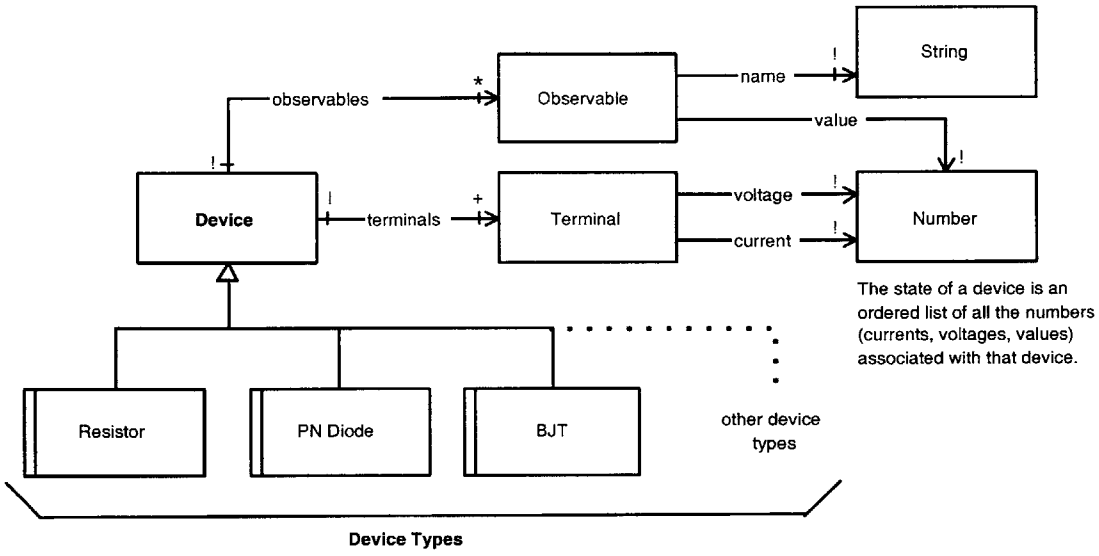


Figure 2-1: Object model showing the relationships among devices, device types and terminals.

A state satisfies a device model if and only if the components of the state satisfy the constraints imposed by the model's equations.[2] Because the model equations mention specific components of a device state, a model is intimately associated with one device type. This corresponds to our intuition that a model describing a pn diode, for example, does not "work" for a resistor nor in general, for a different kind of device.

A device model constrains only the components of the device state that are involved in the model's equations. The observables whose values are determined by a model are the *model observables.* All other observables are left unconstrained; thus, it is not useful to consider their values when working with the particular device model.

Example: Ideal p-n Junction Model of a pn Diode [24]

Consider a silicon pn diode modelled as a one-dimensional semiconductor structure where the doping level abruptly changes from n-type, with a uniform donor concentration N_D, to p-type, with a uniform acceptor concentration N_A. Applying semiconductor physics, we find that there is a region around the junction (the *depletion*

[2] In the concrete representation introduced in note 1, the function $\vec{f}(\cdot)$ representing the device model may be defined such that a given device state $\vec{x}$ satisfies the model if and only if $\vec{f}(\vec{x}) = 0$.

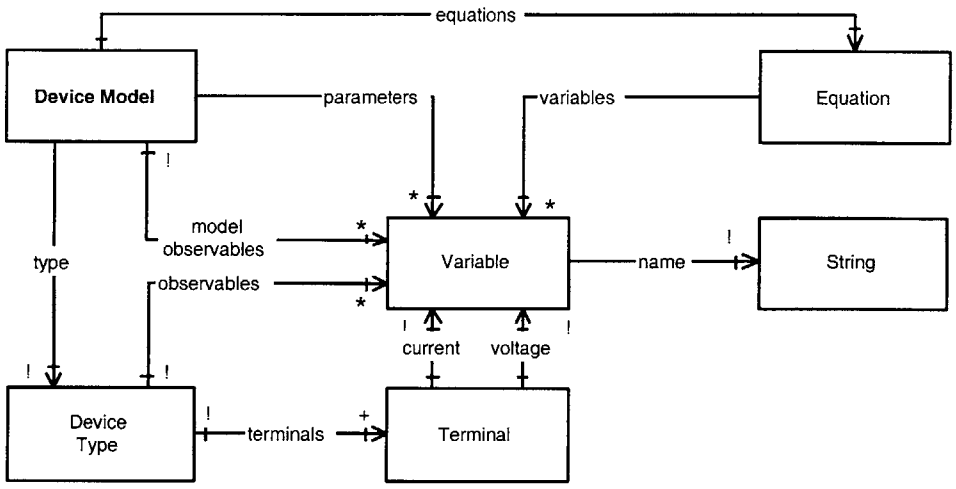


Figure 2-2: Object model showing the relationships among device models, device types and model observables, and parameters. The objects reachable via the "model observables" relation is a subset of the objects reachable via the "observables" relation.

region) where the bulk charge density is non-zero. Deep inside the semiconductor, sufficiently far away from the junction, the charge density falls to zero. If this *quasi-neutral region* is much longer than the minority carrier diffusion length, the voltages and currents at the terminals of the diode are related by:

$$I_1 = A\left(\frac{qn_i^2 D_e}{N_A L_e} + \frac{qn_i^2 D_h}{N_D L_h}\right)\left(\exp\frac{q(V_1 - V_2)}{kT} - 1\right) \quad (2.1)$$

$$I_2 = -I_1 \quad (2.2)$$

In Equation 2.1, V_1 (V_2) and I_1 (I_2) are the voltage and current at the p-type (n-type) terminal, respectively. A is the area of the junction, n_i is the intrinsic carrier concentration, q is the electronic charge, k is Boltzmann's constant, L_e (L_h) is the electron (hole) diffusion length, D_e (D_h) is the electron (hole) diffusion coefficient, and T is the temperature of the diode. The values of q, k are fixed by the laws of nature. Only n_i, L_e, L_h, D_e, D_h, N_D, N_A, A and T may be adjusted. These nine quantities are the parameters.

These parameters are not necessarily independent of each other. For example, the diffusion lengths and the diffusion coefficients above are roughly related to the

doping level in the semiconductor. In practice, Equation 2.1 is often simplified by introducing an empirical parameter I_S (the saturation current) which represents the combined effects of n_i, L_e, L_h, D_e, D_h, N_D, N_A and A. This gives

$$I_1 = f(I_S, T, T_{nom}) \left(\exp \frac{q(V_1 - V_2)}{kT} - 1 \right) \tag{2.3}$$

where T_{nom} is the temperature at which I_S was measured, and f is a function that captures the temperature dependence of I_S.

Formally, the state of a device of type "silicon pn diode" is captured by the vector $[V_1, I_1, V_2, I_2]$. The "ideal p-n junction model" for this device type is defined by Equations 2.4–2.5.

$$I_1 = f(I_S, T, T_{nom}) \left(\exp \frac{q(V_1 - V_2)}{kT} - 1 \right) \tag{2.4}$$

$$I_2 = -I_1 \tag{2.5}$$

In this simple model, I_S, T and T_{nom} are the model parameters. For this example, we chose to not include any model observables, limiting the ideal p-n junction model to the I-V characteristics of the diode. However, the model can be developed further to yield expressions for observables (e.g., dynamic resistance, junction capacitance).

2.1.3 Device Simulator

Conceptually, a *device simulator* is a computer program that takes as input a device model, a sequence of partially-specified device states (including values for the parameters of the device model), and a set of parameter values that control the behavior of the simulator. When the simulator runs, it "fills in" the partially-specified states such that each of the completed states satisfies the device model. The output of the simulator is the sequence of states corresponding to each partial state in the input. We call the input to the simulator a *simulation*, and call the output the result of the simulation.

A simulator implements a device model if it accepts simulations that use the

given model. Most simulators implement a variety of device models; conversely, a device model can be implemented by many simulators. For example, the Level 1 SPICE MOSFET model is supported by Berkeley Spice3F4, and by all other Spice-compatible circuit simulators. The implementors of a device model can choose to give a default value to some parameters in the model, so that omitting the parameter's value is equivalent to specifying the default.

Model Simplifications

Concrete realizations of a device simulator often add additional functions to make the program more convenient to use. When a device simulator is exposed to users through WeblabSim, it is presented with the following features[3]:

- The input sequence of partial states are not specified explicitly. Instead, the user provides control statements that generate the input sequence. This means that the user does not have to specify the inputs to the device in painstaking detail at every point in time. For example, rather than giving all the voltages at 0.1 V intervals from -1.0 V to 1.0 V, the user can simply declare that he wants "the voltage swept from -1.0 V to 1.0 V, in 0.1 V intervals." The next point lists the kinds of commands that can be given to the simulator.

- In generating the input, the simulator can

  - hold a parameter, terminal current or terminal voltage at a constant value (`CONS` function)

  - sweep a parameter, terminal current or terminal voltage over a range of linearly- or logarithmically-spaced values (`VAR1` function)

  - at each point in the parameter, current or voltage sweep described in 2.1.3 above, similarly sweep the value of another such quantity (`VAR2` function)

  - while sweeping the current or voltage at terminal A, set the current or voltage at a different terminal B to a linear function of the quantity begin

[3]Section 3.2 discusses the user's view of these functions.

swept at A (`VAR1P` function)

- The user can specify which currents and voltages he is interested in, and only those variables are downloaded.
- The simulator can evaluate expressions (user-defined functions) which are functions of the currents and voltages in the results, and other defined expressions.

If the device simulator actually does not support all these features, the WeblabSim system emulates those that are missing by processing the input and output of the simulator.

To keep the system simple, we associate each device model with exactly one device simulator. This means that WeblabSim treats the Level 1 MOSFET model implemented by Berkeley Spice3F4 and the Level 1 MOSFET model implemented by WinSpice to be different device models, even though they may be functionally equivalent. This makes it much easier to determine which simulator to use to run a given simulation: we simply use the simulator associated with the device model specified in the simulation. This means that the user's choice of device model while configuring a simulation effectively determines the following: the simulator that runs the simulation, the device model to use, and the default values for any parameters that the user omits.

We avoid issues caused by incompatibilities between different simulator versions by defining different versions of the same simulator to be different simulators altogether. Thus, to the WeblabSim system, WinSpice version 1.05.01 and WinSpice version 1.05.04 are two different simulators.[4]

2.1.4 Device Profile

A laboratory administrator may want to customize device models before presenting them to users. Device models can be quite complicated. For instance, the Level 1

[4]An important implication of this decision is that the set of device models implemented by a given simulator is fixed. Therefore, a simulator can be released as a package containing the code needed to run it, and definitions for all the device models that it supports. Once this simulator package has been loaded, the laboratory administrator should not need to configure it any further.

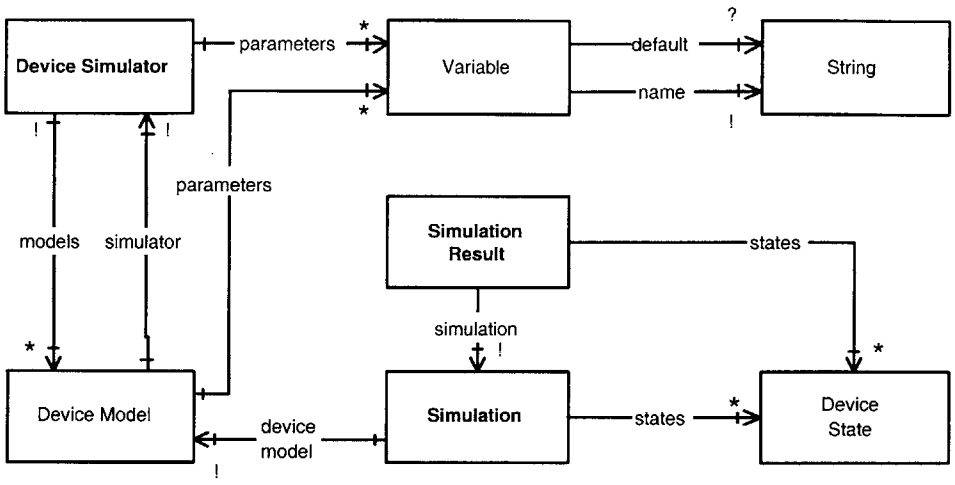


Figure 2-3: Object model showing the relationships among device simulators, device models and simulations.

junction diode model in HSPICE^{TM} has more than 23 parameters [25]. It could be desirable from an educational standpoint to limit the number of parameters that users see at a given time. For instance, if an exercise focuses on the ideal model of a silicon pn diode, students should not have to set parameters like the breakdown voltage or the flicker noise coefficient. Even though the simulator often provides default values for these parameters, the mere presence of these parameters in the device setup dialog clutters the interface and could distract the user from the task at hand.

A *device profile* is a customized device model. By defining a device profile, the laboratory administrator can:

- define a descriptive name (e.g., "ideal silicon pn diode") that will guide users in selecting a profile to use in their simulation
- re-define the default values for model/simulator parameters
- hide model/simulator parameters. Hidden parameters are not presented to the user in the device setup dialog. However, if the administrator specifies a default value, the parameter's default value is still passed to the device simulator.
- tell the system to withhold (omit) certain model/simulator parameters from the simulator. Omitted parameters are not passed to the device simulator at all,

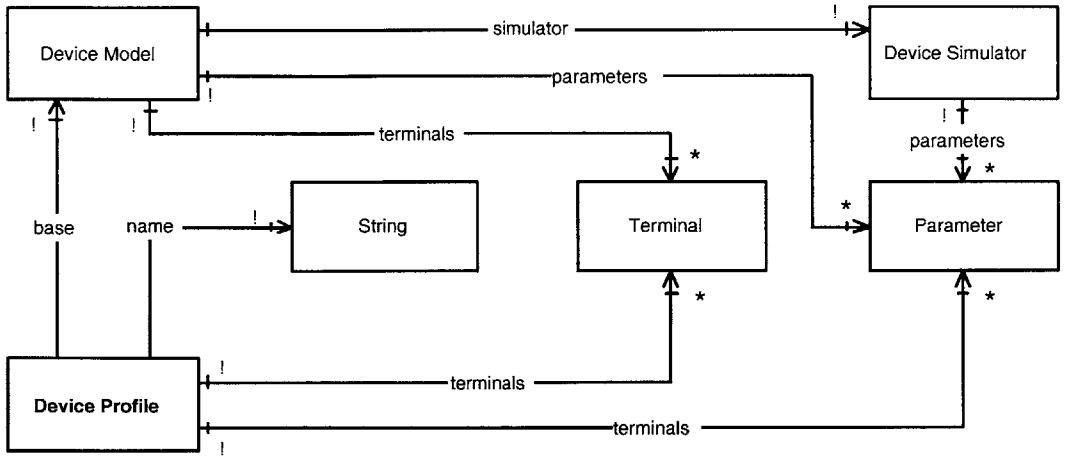


Figure 2-4: Object model showing the relationship between device profiles and device models. The relations of a device profile p are further constrained as follows: $p \rightarrow parameters \subset (p \rightarrow base \rightarrow parameters \cup p \rightarrow base \rightarrow simulator \rightarrow parameters)$ (a device profile can eliminate parameters, but not add new ones), and $p \rightarrow terminals = p \rightarrow base \rightarrow terminals$. (every terminal in the device model must be represented in the device profile).

even though a default value was assigned to them at the device model or device profile level. [5]

- set the maximum number of data points in a simulation involving this device profile

Many device profiles can be defined from a single device model. Supposing the WinSpice v1.05.04 junction diode model is installed, the administrator can create device profiles like "ideal silicon pn diode", "silicon pn diode with series resistance", "silicon pn diode with breakdown", and "germanium pn diode" from this device model by setting the appropriate parameters correctly.

Example: A Germanium pn Diode

The junction diode model implemented by WinSpice v1.05.04 has 15 parameters, nine of which are relevant to the diode's DC characteristics (including temperature

[5] In some simulators, there is no way to explicitly specify what would have been the default value of a model parameter had it been omitted. For example, in the WinSpice simulator, the only way to set the reverse breakdown voltage for a junction diode to infinity is to omit the BV parameter from the MODEL statement.

Name	Description	Units	Default
IS	saturation current	A	1.0×10^{-14}
N	ideality factor	-	1.0
RS	series resistance	Ω	0.01
EG	band gap energy	eV	1.11
TNOM	parameter measurement temperature	°C	27
TEMP	simulation temperature	°C	27

Table 2.1: WinSpice junction diode model parameters that control the DC characteristics of a pn diode.

dependence) [26]. If we focus on a semiconductor diode and ignore breakdown, only the parameters in Table 2.1 need to be considered.

A simple device profile that emulates the DC characteristics of a germanium pn diode can be defined by:

1. setting the band gap energy to 0.661 eV (`EG=0.661`),
2. increasing the default value of the saturation current to reflect the higher intrinsic carrier concentration and larger carrier diffusion coefficients in germanium (e.g., `IS=1.50E-5`),
3. setting `TNOM=27` and documenting that whatever value the user provides for `IS` is the saturation current as measured at 27 °C, and
4. hiding `EG`, `TNOM`, and the parameters not present in Table 2.1, so that the user cannot change their values.

This example illustrates the differences between the concepts of a device, a device type, a device model and a device profile. A germanium pn diode is a device. Its characteristics suggest that we classify it as a "pn diode", which is its device type. PN diodes can be modeled as an "ideal p-n junction", as described in section 2.1.2; we can use this device model for the germanium diode by giving appropriate values for the model parameters. If users had to provide these values every time they wanted to simulate a germanium diode, they will soon find this process tedious and error-prone. To make it easier for WeblabSim users, we can define a device profile named "germanium pn diode" that has the relevant parameters preset to the correct values.

2.2 Design Goals

According to MIT professor Daniel Jackson, A system is well-designed if it has the following key properties [27]:

Extensibility The design must be able to support new functions. A system that is perfect in all other respects, but resists the slightest change or enhancement, is useless.

Reliability The system must behave reliably. The system must not crash or corrupt data, and must perform its functions correctly, as anticipated by the user.

Efficiency The resources consumed by the system must be reasonable. This metric depends on the application's context: a server program that runs on powerful machines may depend on the availability of more resources than a client designed to run on a variety of platforms.

Applying these principles to WeblabSim, an online simulator intended for use in an academic environment, results in the following design objectives. Each objective is driven by a user requirement from the problem domain, but the statement is informed by considering the design principles just described.

- At the minimum, the system must support the same modes of user interaction as WebLab. WebLab has been running for six years now [18]. Semiconductor parameter analyzers, the instruments on which WebLab was based, have been around for even longer. It is safe to assume that the ways in which these tools can be used generally satisfy the needs of the engineering community. Therefore, their features would be a reasonable base to start from when designing WeblabSim's own capabilities.

  At the same time, the fact that WeblabSim is a simulator means that it can do some things that WebLab currently does not do, or perhaps cannot do. For example, the current version of WebLab cannot measure the gate-source capacitance of a MOSFET. In WinSpice, this capacitance is easily obtained

by printing the value of the cgs output parameter [26]. The designed system should be flexible enough so that these extra features can be exposed to the user in future versions of WeblabSim.

- The system must support simulators of varying complexity. This thesis describes a deployment of WeblabSim that targets two different simulators: WinSpice v1.05.04 [1], and a simple device simulator that uses the Maple™ symbolic algebra tool [28] to solve device equations. We expect the system's framework to be general enough that other simulators can be used as well.

  Apart from making WeblabSim extensible, this requirement also makes the system scale to meet the demands of its users and deployers. Future adopters of the technology should be able to customize their online simulator to their needs. A simple simulator may be all that a small-scale deployment can support, whereas a larger university may want to use a more powerful circuit simulator like HSPICE™.

- The system must scale to support a large user base. WebLab, the current online remote microelectronics laboratory, has served over 1,900 users since its inception [18], and is expected to handle over 1000 users in the Fall 2004 term alone [22]. Serving such a large number of users can quickly consume system resources. It may also compromise reliability by exposing obscure bugs. This problem is especially important to WeblabSim, as its functionality is implemented entirely in software. A single ill-behaved component could be disastrous for the entire system.

- The system must be modular, so that we can build on existing components when this is possible. Since WebLab's inception, several other online laboratories have been developed at MIT. Our group has observed, in conjunction with the other groups developing online laboratories, that there are many tasks common to applications of this kind. These included user management, authentication, authorization and storage of experiments and results. We decided that for

WeblabSim, we would use (as much as possible) existing generic components to handle these tasks.

2.3 The iLab Batched Experiment Architecture

We satisfied the final design goal above by adopting the iLab Batched Experiment Architecture (BEA) [22]. The BEA is a system design blueprint developed by the MIT iLab project [29] to support a scalable community of online experiments for use by students at multiple institutions.

The iLab design is a three-tier architecture with the following layers (Figure 2-5):

1. The student's *client application*, by which the user accesses the online laboratory. Client applications may be dynamic web pages, browser applets or downloadable applications. The user defines an experiment through the client's user interface. The client then synthesizes an experiment specification, which it submits to the service broker. Later on, the client asks the broker for the results of the experiment, and displays them to the user.

2. A set of generic middleware components called *service brokers*, which mediates between the client application and the laboratory. The service broker provides shared common services (authentication, authorization, shared storage, etc.) to several clients and laboratories. Initially, the service broker authenticates the user and determines which laboratories he has access to. When the user has started using the client, the service broker acts as a proxy between the client and laboratory server. Among others, it has methods for passing through the laboratory configuration, experiment specification and experiment results.

3. The *laboratory server*, which executes the specified experiments and notifies the service broker when the results are ready.

The methods that comprise the application programming interfaces (APIs) between the client application and the service broker, and between the service broker and the laboratory server, are listed in Appendix B.

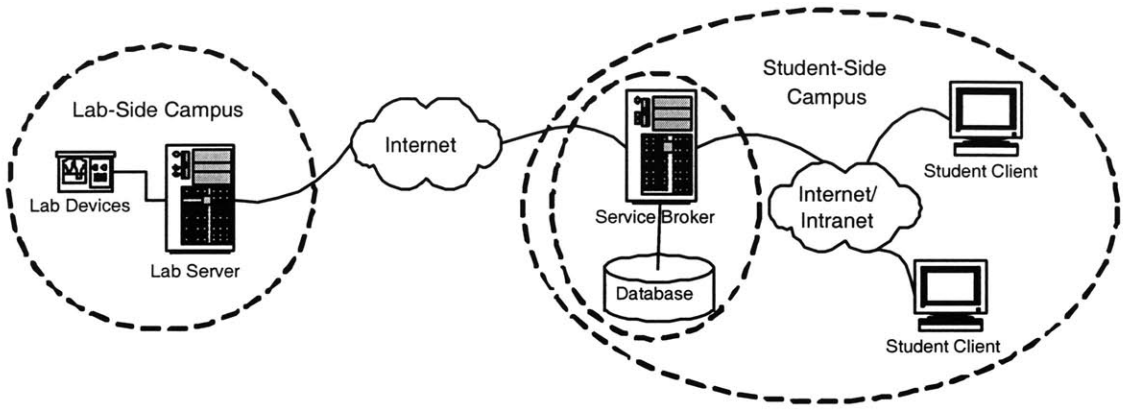


Figure 2-5: Topology of the iLab Batched Experiment Architecture (from [22])

The iLab blueprint offers two primary insights. First, user management issues, policies related to authentication and authorization, and the storage of experiment specifications and results are delegated to the service broker, instead of being handled directly by the laboratory server. This indirection results in a better distribution of communication and user management throughout the network, where it was once concentrated at the laboratory server.

Second, the BEA calls for the use of SOAP-based web services [30, 31] between the client and service broker, and between the service broker and laboratory server. This allows the architecture to make minimal assumptions about the platforms used by students, experiment implementers, or universities—as long as the components conform to the BEA specification, they ought to be interoperable. But apart from making the overall system platform-neutral, using web services promotes logical and physical decoupling between pieces of the system. The different services run as separate processes—in fact, they can even (and usually do) run on separate machines—making it easier to render components resilient to errors in other parts of the system. For instance, by strictly validating the input to the laboratory server and locating it on its own dedicated machine, we can keep the server running even if one service broker crashes, or worse, begins producing garbage.

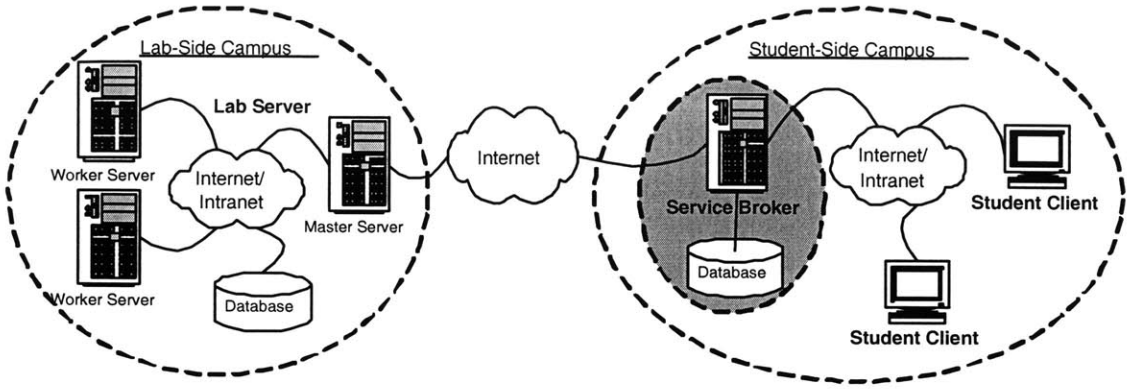


Figure 2-6: Topology of the MIT Device Simulation WebLab

2.4 Overview of the WeblabSim Architecture

Figure 2-6 shows the WeblabSim system as implemented through the iLab Batched Experiment Architecture. The overall system is divided into the client application, the service broker and the laboratory server, as prescribed by the iLab blueprint. But because WeblabSim is a simulator, not an online laboratory, the design departs slightly from that of Figure 2-5. The new topology meets the needs of system that is implemented completely in software, while taking advantage of its strengths.

2.4.1 Client Application

The client application is a Java browser applet that communicates to the service broker through web service calls. The applet's user interface, design and implementation will be described in Chapter 3.

2.4.2 Service Broker

The service broker, marked by a grey oval in Figure 2-6, was developed by the iLab project as a generic component that can be used for various online laboratories. Although the system as a whole depends on the services of the broker, the service broker itself is completely independent of WeblabSim. We shall not discuss the implementation of the service broker in this thesis. However, in the succeeding chapters we will describe the services provided by the broker when relevant to the client application

or the laboratory server. [22]

2.4.3 Laboratory Server

Instead of being single entity, as prescribed by the generic iLab BEA, in WeblabSim the laboratory server is divided into several parts: a *master server*, several *worker servers*, and a database that holds the state of the system. The pieces are logically separate from each other, but together, they present a "virtual laboratory server" to the service broker.

The master server implements the laboratory server web service interface required by the iLab architecture. It acts as a façade, making the collection of workers, master and database behave like a single laboratory server from the service broker's point of view. Thus, the master server is responsible for giving information about the WeblabSim system, reporting the status of the server, receiving experiment specifications and reporting the experiment results. The master server validates the incoming experiment specification and determines the order in which the simulations will be run. When a job is selected for execution, the master selects which worker server will perform the simulation. The selection strategy is configurable. The current implementation uses a variety of simple selection algorithms (round-robin, random, polling [32]), but one can imagine more sophisticated strategies that do explicit load-balancing or license management.

The worker servers are the components that actually run the simulations received by the laboratory server. The workers implement the generic simulator interface defined in section 2.1.3. The worker server passes the input through a series of filters which simplifies it by implementing in software the features of the generic simulator that are not supported by the particular simulator specified (targeted) in the input. The worker then invokes the target simulator with the simplified input, perhaps even repeatedly, if necessary. Another chain of filters casts the output of the simulation into the form expected by the rest of the system. User-defined functions are evaluated as part of this post-processing.

This distributed design makes it relatively easy for WeblabSim to cope with a

swelling user base by adding more computers to the laboratory server. Each of the three types of components is replicable, should it become the bottleneck. Moreover, the three-way split between components that store data, run simulations and manage jobs lets the administrator focus on the component which is the actual bottleneck in the system. For most WeblabSim installations, running simulations is the task that requires the most computation, so the administrator will likely have to add more worker servers first, before needing more data stores or master servers.

The following three subsections give more details on how the components of the laboratory server can can be configured to cope with increased load.

Database If a solid, enterprise-grade database system is used as a data store, the database is unlikely to become the bottleneck in the system. But should it ever be the weakest link, database replication can be used to store copies of the data on multiple computers (full replication), thereby speeding up database queries [33]. This technique will work well with WeblabSim, as the system accesses data regularly, but updates it infrequently.

Worker server By design, the simulation service provided by the worker is stateless, meaning the simulation can be performed based only on the input provided by the master. Also, the simulation can be run repeatedly, producing the same results each time (i.e., the simulation operation is idempotent). This combination of properties make worker servers particularly easy to replicate. It does not matter to the master *which* worker server performs the simulation on its behalf, as long as *some* worker does it, so the master simply has to pick one. And whereas one usually needs special strategies for detecting and handling changes in the worker set (e.g., failures, workers leaving, workers joining), the idempotent property of simulation leads to a simple change recovery protocol: If a worker doesn't finish a simulation, try again with another worker server.

It is relieving to note that replicating worker servers is relatively easy, because the ability to replicate workers is particular important. A simulation could potentially require a large amount of computing power. Even if the computation

needed for any one simulation is minimal, the combined impact of hundreds of users running simulations at the same time could overwhelm a single computer. With more than one worker, the system can spread the computing burden over a number of units, reducing the load on each one to a more manageable level.

Master server If the master server cannot cope with the volume of users, the administrator can add more master servers to the system. Because all the masters use the single (possibly replicated) database as their data store, they effectively act as multiple views of the same laboratory server. The façade of a single laboratory server is completed by (among other ways):

- hiding the multiple masters behind a load balancer that assigns each server the same effective Internet Protocol (IP) [34, 35] address, or
- configuring the Domain Name Service (DNS) [36] server to randomly return one of the masters in response to a query for the domain name of the laboratory server (e.g., http://simulator.example.org).

Service brokers accessing the laboratory server through this IP address or domain name will then be routed transparently to one of the master servers.

2.5 Summary

This chapter presented the overall philosophy behind the design of the MIT Device Simulation WebLab. We motivated the system architecture by analyzing the problem, and then enumerating the particular qualities we wanted the design to have. These qualities were then developed into concrete goals by linking them with particular user requirements. We introduced the iLab Batched Experiment Architecture, the blueprint on which we based our system design. The chapter ended with a brief overview of the components that comprise the WeblabSim system.

The next two chapters describe the implementation of WeblabSim's domain-dependent components in greater detail. Chapter 3 discusses the client applet, and Chapter 4 focuses on the laboratory server.

Chapter 3

Client Applet Implementation

The client applet is the primary user interface of the MIT Device Simulation WebLab. A laboratory client is the specialized, domain-dependent piece of software that the user directly interacts with in order to create experiment specifications, submit them for execution (via the service broker), and analyze the results. Not everything the user does is performed through the client; some of it, such as the initial login process, is done through a standard web browser before the client is even loaded.

This chapter describes the WeblabSim client applet. It starts by briefly introducing the WebLab client, the forerunner of the WeblabSim program. The next section describes the applet's user interface. We then discuss the details of the WeblabSim client, with emphasis on the changes and improvements that we made to the WebLab code. The chapter concludes with an estimate of the system resources that will be consumed by the client applet.

3.1 WebLab Client Applet

The WebLab applet is the product of years of experience obtained while using WebLab in microelectronics classes at MIT and other institutions. The initial version of the WebLab applet was based on the user interface of the HP4155B Semiconductor Parameter Analyzer. In 2002, Victor Chang and Yifung Lin developed a new version of the applet that provided a more graphical "feel" [37, 21]. Instead of a channel

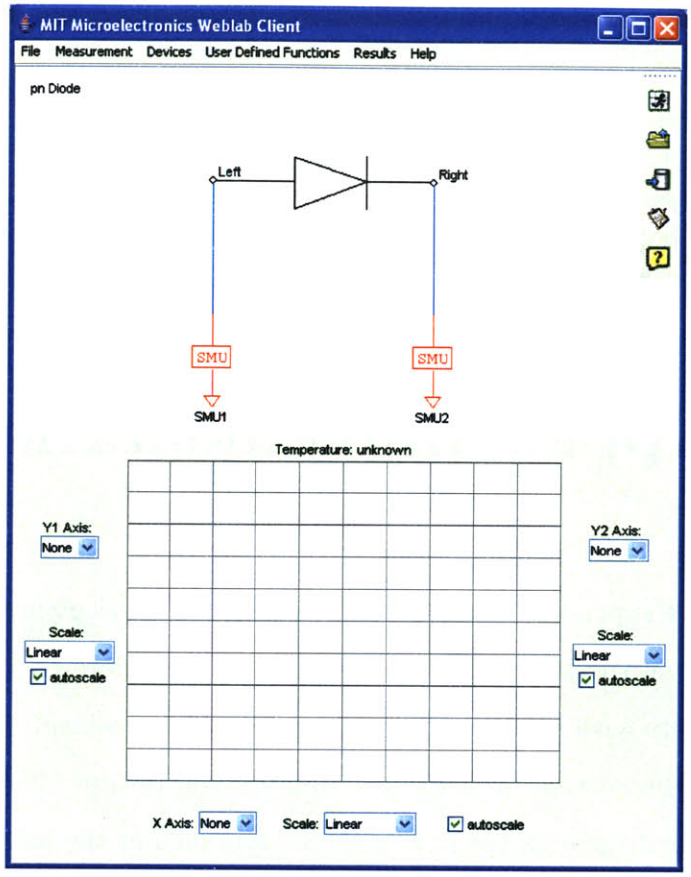


Figure 3-1: User interface of the MIT Microelectronics WebLab client applet, Graphical Version 6.0

definition panel consisting of a series of text input boxes, the new interface uses an image of the device under test. The new interface was well-received by WebLab users.

Moving to the iLab Batched Experiment Architecture precipitated a redesign of the client applet's internals. David Zych implemented the current version of the WebLab applet, which was part of the version 6.0 release of WebLab [22]. The WebLab 6.0 applet has the same graphical look and feel, but the user interface has a streamlined design that makes better use of screen space (Figure 3-1). Of course, the internals of the applet are completely different, primarily because the iLab BEA radically changed the communication patterns between the client and the laboratory server.[1]

[1]Compare the descriptions of the old architecture in [37] and [21] with the iLab BEA described in [22].

3.2 User Interface

A user begins using WeblabSim by logging on to a site that provides access to the service and downloading the client applet. Figure 3-2 shows the applet right after it has been started. The user interface is divided into two parts. The upper half is the *setup panel*, and is used to specify the simulation to be performed. The *results panel*, located in the lower half of the window, graphs the results of the simulation.

There are four menus. The Setup menu contains functions for creating a new setup, for loading or deleting a saved setup, and for saving the current setup. The menu also lets users enter user-defined functions (§3.2.1) and submit the current setup for execution. The Devices menu presents a list of device profiles (§2.1.4) that are available for simulation. Exactly one device model can be active at a given time. The Results menu has items that let the user view the raw simulation data or download it to his computer for further analysis. Finally, the Help menu provides information about WeblabSim and the current version of the client applet. Some of these functions are also accessible through a toolbar located on the right edge of the setup panel or via predefined shortcut keys. See Table 3.1 for details.

3.2.1 Configuring a Simulation

To begin defining a simulation, the user selects a device profile from the Devices menu. The setup panel changes to reflect the kind of device represented by the model (e.g., pn diode, n-type MOSFET) and how the terminals of the device are connected to test units. Hovering the mouse over image of the device displays a small window that lists the model parameters and their current (default) values. This is illustrated in Figure 3-2.

Setting Device Model Parameters

Clicking on the device brings up the *device setup dialog*, shown in Figure 3-3(a). Each line of the device setup dialog has information about one model parameter: its name, a short description, its measurement units, and its current value. At the end of the

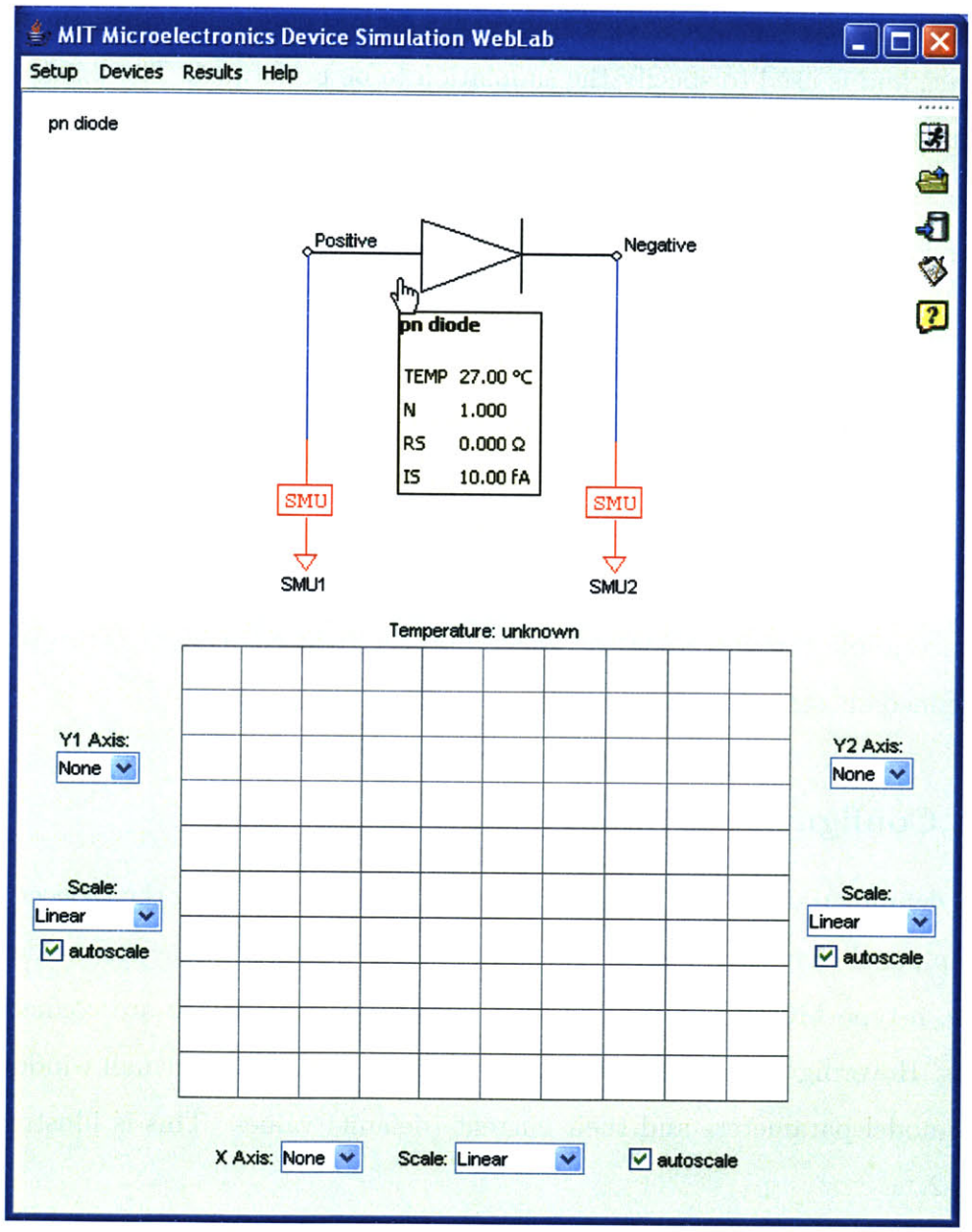
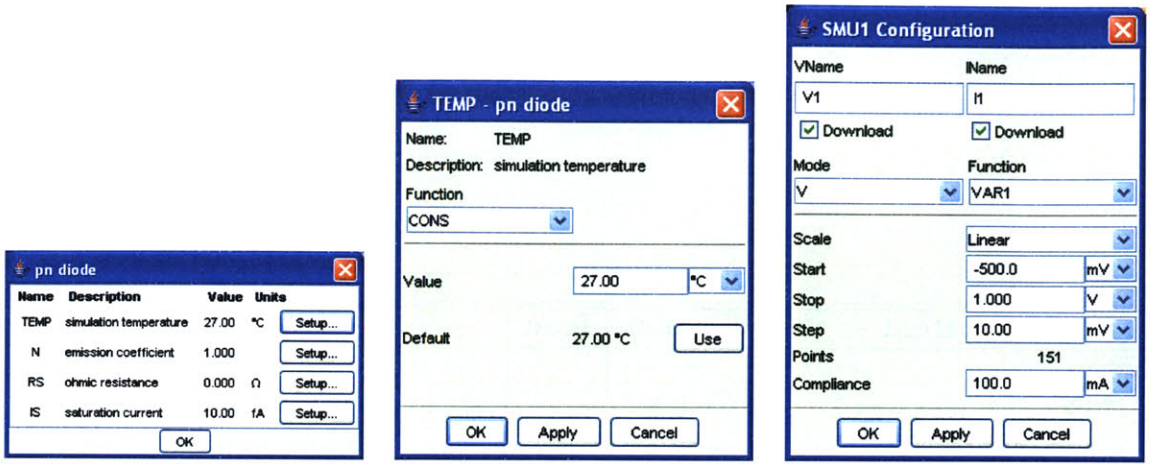


Figure 3-2: User interface of the MIT Device Simulation WebLab

Menu Item	Shortcut	Icon	Description
Setup			
New	Ctrl-N		Creates a new setup
Open	Ctrl-O		Loads a saved setup
Save	Ctrl-S		Saves the current setup
Delete			Deletes a saved setup
User-Defined Functions			Shows the dialog for entering user-defined functions
Run	F5		Sends the simulation to the server for execution
Exit			Closes the WeblabSim applet
Devices			
device profile name			Sets the device profile as the active profile. This is the profile that will be used when the simulation is submitted.
Results			
View			Opens a window showing the raw simulation data
Download			Downloads the simulation results to the user's computer
Help			
Lab Info			Displays a URL with more information about WeblabSim
About WeblabSim			Shows information about the applet

Table 3.1: Menu and toolbar items in the WeblabSim applet.



(a) Device setup (b) Parameter setup (c) SMU setup

Figure 3-3: Dialogs used to configure a simulation

line is a button for changing the parameter's configuration. When the user clicks on this button, the *parameter setup dialog* shown in Figure 3-3(b) is displayed.

The parameter setup dialog controls the behavior of a model parameter during the simulation. The dialog displays the name and a description of the parameter being configured. Through the "Function" setting, the user can choose to hold the parameter at a constant value (CONS), sweep it over an interval through a linearly- or logarithmically-spaced set of points (VAR1, VAR2), or omit the parameter altogether (OMIT).[2] Selecting a function changes the dialog to reflect options specific to the chosen function. For example, because the user has selected a CONS function for the parameter in Figure 3-3(b), the dialog has fields for entering the value of the parameter, and a button that resets the parameter to its default value.

Programming Source and Monitor Units

The terminals of the device are connected to programmable *units*. There are three kinds of units. A Voltage Source Unit (VSU) supplies a defined voltage to a terminal. A Voltage Monitor Unit (VMU) measures the voltage at a terminal but draws no current from it. A Source/Monitor Unit (SMU) functions in one of three modes:

[2]Omitting a function means that it is not sent to the server as part of the simulation specification. See section 2.1.4 for a discussion of why this is sometimes necessary.

voltage source/current monitor, current source/voltage monitor, and connection to ground. Units are configured by clicking on their representation onscreen, which brings up the *unit setup dialog*, an example of which is shown in Figure 3-3(c).

The user sets up the SMU for the simulation through the unit setup dialog. The dialog lets the user name the voltage and current into the terminal the unit is connected to, and specify whether each should be downloaded as part of the simulation results. The "Mode" option selects which of its three modes the SMU should operate in; the "V" selection instructs the SMU to operate as a voltage source/current monitor. The user configures the time-evolution of the source by setting the "Function". As with the model parameters, VAR1 or VAR2 means the source is stepped through a series of values, and CONS means that the source is kept at a constant value throughout the simulation. The SMU function can also be configured as VAR1P, which provides a linear function[3] of the value supplied by the SMU unit configured as VAR1.

Entering User-defined Functions

The user can run calculations on the results of the simulation by submitting *user-defined functions*. Selecting the menu item Setup | User Defined Functions (or clicking on the equivalent toolbar button) displays a window that lets the user enter, edit and delete user-defined functions. Functions may be defined in terms of currents, voltages, model parameters, and other functions. However, recursive and mutually-recursive definitions are not allowed.

Appendix A defines the expression language used for user-defined functions in WeblabSim. The syntax and semantics of the language is based on the functions provided by the Agilent HP4155B Semiconductor Parameter Analyzer, which is at the core of the WebLab system.

The Configured Simulation

The setup panel of Figure 3-4 shows an example of a setup configured as follows:

[3] A linear function $y = f(x)$ is defined by a ratio and and an offset, such that $y = ratio \cdot x + offset$.

- The device model is an ideal pn diode with a series resistance. The model has four parameters: the temperature (TEMP), the saturation current (IS), the ideality factor (N) and the series resistance (RS).

- The model parameters have been set to constant values. TEMP = 27°C, IS = 10 fA, N = 1, and RS = 0 Ω.

- SMU1, which is connected to the positive terminal of the diode, is acting as a voltage source. The voltage and current are named $V1$ and $I1$, respectively, and will be downloaded. The voltage will be swept from -0.5 V to 1.0 V in 10 mV increments.

- SMU2, which is connected to the negative terminal of the diode, is acting as a direct connection to ground.

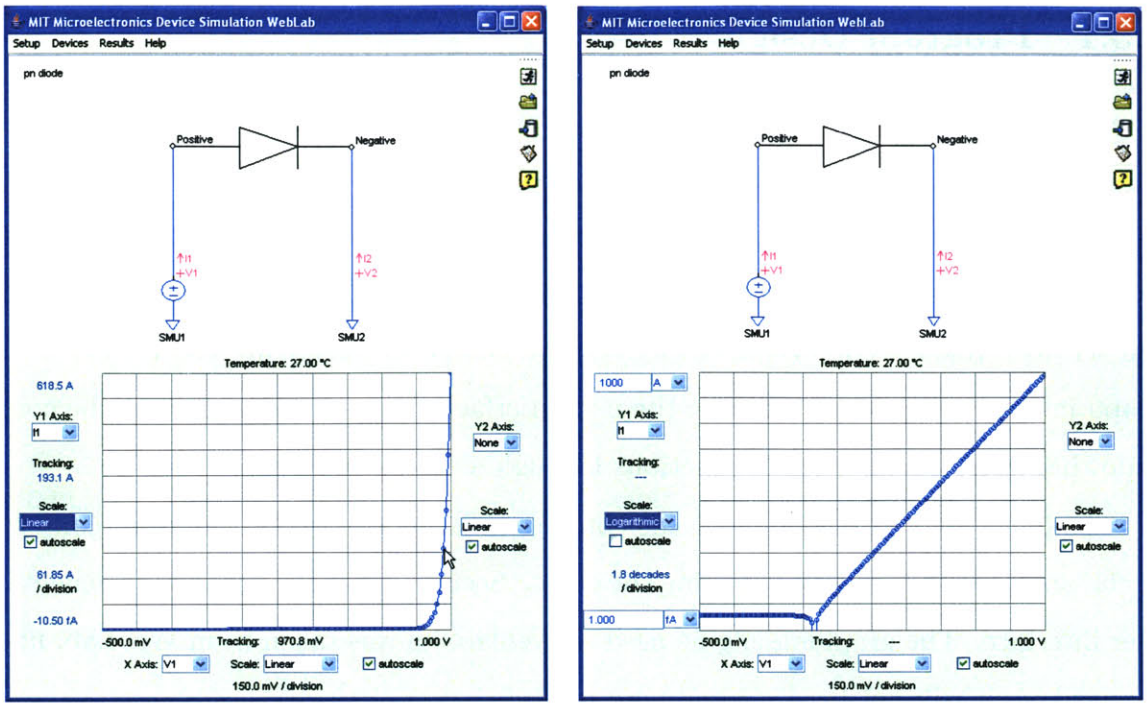
- No user-defined functions.

3.2.2 Running the Simulation

To run the simulation described by the current setup, the user can either select the menu item Setup|Run, press the F5 key, or click on the equivalent toolbar button. This displays a progress dialog that disappears when the simulation is complete. If the simulation completes successfully, the results are graphed in the results panel. Otherwise, the program displays an error message.

3.2.3 Graphing the Results

When a simulation completes successfully, the results (if any) are graphed in the results panel. The user can select up to two variables (Y1 and Y2) to plot against another variable on the x-axis. The scale of each axis can be either linear or base 10 logarithmic. The user has the option to manually set the minimum and maximum values for each axes, or to have the program do this automatically by selecting "autoscale". Figure 3-4 shows the results of the simulation described above, with I1 on



(a) Linear

(b) Semi-logarithmic

Figure 3-4: The WeblabSim applet displaying the results of a simulation. The client is graphing I1 against V1. Note the exponential increase in current in forward bias and the current saturation in reverse bias. This is consistent with the ideal pn junction model.

the y-axis plotted against V1 on the x-axis. Figure 3-4(a) is a linear plot; Figure 3-4(b) shows the same data, but with the y-axis on a logarithmic scale (semi-logarithmic plot).

The user can also view the data as raw lists of numbers by selecting Results | View from the menu. He can also download as a comma-separated values (CSV) file through the Results | Download menu item.

3.3 WeblabSim Client Applet Implementation

The WeblabSim client applet was developed based on the WebLab client. The WebLab source code was forked on March 30, 2004. From then on, development on the two applets proceeded independently.

3.3.1 Program Design

The applet is divided into four parts: data/core logic, service broker interface, graphing engine, and the user interface. The classes that comprise the data/core logic part keep track of the applet's state and implement the business logic behind the application. The service broker interface abstracts away the details of communicating with the service broker. The graphing engine is in charge of displaying the results as a graph in the results panel. Finally, the user interface classes is a veneer over the core logic that presents the client's functions through a graphical user interface.

The following sections describe the core logic and service broker interface components, and the data model underlying the client. Section 3.2 above presents the client's user interface. The graphing engine used in WeblabSim was taken from WebLab, unchanged, so it will not be discussed here further.

Data/Core Logic

The data/core logic classes maintain the applet's data model and implements the business logic behind the application. These objects are also responsible for making sure that the applet submits only valid simulation specifications. Although the service broker exposes a Validate method for precisely this purpose, the validation logic in the applet can typically provide immediate feedback to the user, which helps him locate and fix problems.

Service Broker Interface

The service broker interface is a layer of code that exposes the API between the client and the service broker. The methods in the interface were expressed as a Java interface, two implementations of which were developed. The first one is a lightweight class containing pre-determined data, which was used while developing, testing and debugging features. The second is the one used in the deployed applet. It uses the Enhydra kSOAP 1.2 library [38] to communicate with the service broker through SOAP over HTTP [31].

WeblabSim is very liberal in its support for characters in other languages. Our general policy is to support all the characters in the Unicode specification [39]. This is quite useful for model parameters, which often have greek names derived from their symbols. For example, the parameter controlling the carrier mobility in a MOSFET is called "mu", for its symbol μ. The ability to display characters which are not in the ASCII set [40] is also useful for displaying measurement units, which also often have greek symbols (e.g., Ω, μm). Unfortunately, the version of kSOAP we were using permitted programs to use only the default character encoding of the underlying platform. To remedy this, we had to modify the org.ksoap.transport.HttpTransportSE class so that when it receives a SOAP message, it detects the character encoding used by the sender, and interprets the incoming bytes properly.

3.3.2 Data Model

The data model of the WeblabSim applet (Figure 3-5) was derived from the WebLab client. In turn, the WebLab model was based on the implicit model behind the HP4155B Semiconductor Parameter Analyzer, which is the hardware that runs WebLab. Thus, the primary class is Analyzer, which represents the HP4155 unit in the WebLab system. The analyzer has four Source/Monitor Units (SMU), two Voltage Monitor Units (VMU) and two Voltage Source Units (VSU). The number of units in each set are fixed when the analyzer is created; just like the real instrument, the Analyzer cannot lose ports or acquire new ones during normal operation. The ports in each set are numbered, starting from one.

A VMU has a name for the voltage that it measures. A VSU has a name for the voltage that it provides, and a function (SourceFunction) that describes the time evolution of the voltage. An SMU has names for the voltage and current that it provides/measures, a mode that determines how it will operate (current source, voltage source, or ground connection), and a SourceFunction that indicates how the value supplied by the source changes during the experiment. All three kinds of ports have boolean flags that indicate whether the values supplied and/or measured should be downloaded to the client applet.

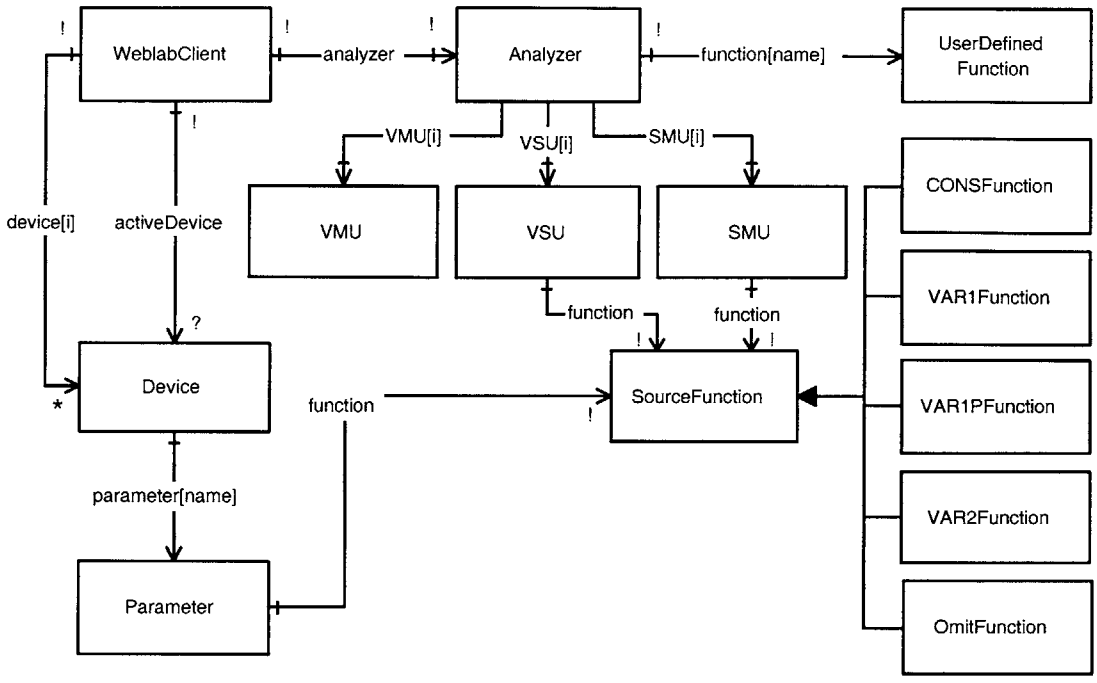


Figure 3-5: Internal state of the MIT Device Simulation WebLab client

We found that the three kinds of SourceFunctions present in the WebLab applet—VAR1Function, VAR2Function, VAR1PFunction, and CONSFunction—were insufficient to specify all the possible behaviors of a parameter. Like the port of an analyzer, a parameter can be held constant or swept through several values. But unlike an analyzer port, a parameter can also be omitted from a simulation altogether.[4] A new kind of source function, OmitFunction, was added to the system to cover this possibility.

The analyzer also keeps track of user-defined functions as mappings from function names to instances of UserDefinedFunction that hold their definitions. A user-defined function has a name, units, a body, and a boolean flag that tells the system whether or not the value produced should be downloaded.

A device, represented by a Device object, has a device ID, a type (which corresponds with the notion of device type in section 2.1.1), a name, a description, and a URL that gives the location of its image. For validation purposes, the device keeps track of how many terminals it has, and the maximum number of data points that a

[4]Section 2.1.4 explains why this behavior is needed in certain situations.

user of this device may measure.

The device also has a list of parameters, which are represented by instances of the Parameter class. Parameters have a name, units, a description, and a boolean flag indicating whether or not they are required. Optionally, they can have a default value specified for them. And much like the ports of the analyzer, they also have a function that controls their value during the simulation.

The WeblabClient class, representing the abstract notion of a "client program", keeps track of the available devices, and the currently-selected device (the "active" device). It also holds a copy of the last experiment's results. This copy is used to draw the graph in the results panel. It is also used when the user asks to see the individual data points in list, and when the results are saved to disk.

3.3.3 Input/Output Format

Appendix D details the formats used by the WeblabSim client for its laboratory configuration (§D.1), experiment specification (§D.2) and experiment results (§D.3). For the most part, these formats closely parallel their WebLab equivalents, whose specifications are listed in Appendix C. However, because of the slightly different data model, some modifications were necessary. The following sections summarize the differences between the documents used by the two applets.

General Changes

- In the root element of all three documents, the (fixed) value of the lab attribute is now "MIT Device Simulation WebLab", where before it was "MIT Microelectronics WebLab".

- The specversion attribute of the root element is now named version, and its value has been increased to "1.0-b1", the first beta version of release 1.0. Future versions should follow the [major].[minor]–[tag] naming convention, so that the components of the system can assess their compatibility with the document.

Tagged releases There must be no breaking changes between different tagged releases with the same major and minor version numbers.

Minor releases A component that can interpret a document with version number x.y–tag must be able to process a document with the same major version. If document has a lower or equal minor version, no information should be lost during the conversion. Otherwise, information may be lost, but no errors should result.

Major releases A component that can interpret a document with version number x.y–tag may or may not be able to process a document with a different major version. If it cannot understand the document, the component must reject it.

Laboratory Configuration

- The maxVoltage and maxCurrent elements were eliminated.

- The device element accepts zero or more parameter elements. If present, these elements are located after the terminal elements, but before maxDataPoints. The parameter elements have no content, but take the following attributes: name, the name of the parameter; units, the units in which the parameter is measured; description, the parameter's description; default, a default value for the parameter (optional); and required, which indicates whether or not the parameter must be specified (true or false, defaults to false if omitted).

Simulation Specification

- The root element is now named simulationSpecification.

- The deviceID element has been eliminated. The device element takes its place. The resource ID of the device profile to use in the simulation is specified in the resource attribute.

- The body of the device element contains zero or more parameter elements. It has one attribute, name, that gives the name of the parameter that it controls. The body contains a function element, which has been adopted (unchanged) from the WebLab experiment specification. Omitted parameters are simply absent from the simulation specification.

3.3.4 User Interface

The following sections describe the ways in which the WeblabSim client applet differs from its predecessor, the WebLab client. The biggest changes were new dialogs and features added to enable the functions added by WeblabSim. There were also a few minor improvements to the menu structure, and the applet's ease with which the applet's interface can be localized in the future.

Device and Parameter Setup Dialogs

The changes in the data model of the client made it necessary to modify the applet's user interface. The applet must enable the user to modify the values of the new data structures. Therefore, we introduced the device setup dialog (Figure 3-3(a)) and the parameter setup dialog (Figure 3-3(b)). Section 3.2.1 explains how the user examine and set device parameters through these dialogs.

OMIT Function

The new OMIT function is potentially dangerous, as using it with a required parameter makes the simulation specification invalid. Also, because OmitFunction is a subclass of SourceFunction, it could potentially be used to configure a VSU or an SMU. This is clearly undesirable, as the user should not be allowed to "omit" a terminal of the device.

To prevent this from happening, the user interface has the following policy for using the OMIT function:

1. OMIT cannot be specified as a function when configuring an SMU or a VMU.

2. When configuring a parameter, OMIT is enabled only if the parameter is not required. That is, in the laboratory configuration, the parameter's required attribute must either be false or unspecified.

3. The applet tries to avoid assigning OMIT as the default function for a parameter. If a default value is specified, the default function is a CONS function that supplies the default value. If a default value is not specified, but the parameter is required, the applet uses a zero-valued CONS function. The default function is OMIT only when there is no default value given for the parameter and the parameter is optional.

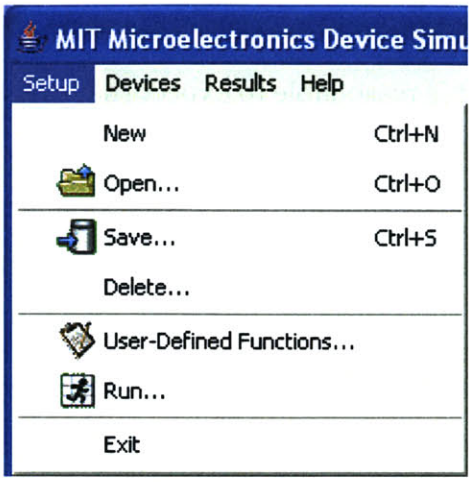
Menus

The WebLab applet has six menus. The File menu has functions to reset the current setup, load a saved setup, delete a saved setup, save the current setup and exit the applet. The Measurement menu has one item, to run the measurement. So does the User Defined Functions, whose only menu item shows or hides the user-defined functions dialog. The Results, Devices, and Help menus are identical to the WebLab menus with the same names (see Table 3.1).

As described in section 3.2 above, the WeblabSim applet has only four menus: Setup, Devices, Results and Help. The Setup menu, located where the File menu used to be, contains the items that used to be in the File, Measurement and User Defined Functions menus. This is a better use of menus, as it groups all the functions relating to the simulation setup into one appropriately-named menu. Instead of searching through three different menus for the actions they can perform on the current setup, the user need only look in one place, the Setup menu.

While we were making changes to the applet's menu structure, we also made the interface more accessible by defining menu hotkeys (e.g., Alt-F for the File menu) and keyboard shortcuts (e.g., Ctrl-N for File|New). These relatively minor changes made the interface much more accessible to a user who cannot or will not use a mouse.

(a) WebLab File menu



(b) WeblabSim Setup menu

Figure 3-6: Comparison of corresponding menus in the WebLab and WeblabSim client applets

Localization

We have also externalized all the strings that are used to build the applet's user interface. Externalization is the process of moving strings from the code to an external properties file. Instead of referring directly to the string, the program retrieves the string from the properties file using a key. For instance, instead of using the string `"File"` directly in the source for the MainFrame class, we:

1. created a file named messages.properties and placed it in the same package as MainFrame,

2. inserted the line `MainFrame.file = File` in the messages.properties file, and then

3. replaced all uses of the string `"File"` with the method call `Messages.getMessage("MainFrame.file")`, which retrieves the string corresponding to the key MainFrame.file from the file named messages.properties.

Externalization makes it easier to localize user interfaces. If we have versions of the messages.properties file with strings in different languages, we can have the Java system automatically select the correct file to use based on the user's locale. The user can then work with the applet whose interface is in the vernacular [41].

WeblabSim is a resource deployed on the internet, a medium with no native language. It is reasonable to expect that eventually, individuals who do not speak English will somehow come across our system and want to use it. MIT's Open CourseWare (OCW) [42], another one of MIT's online educational resources, has already become so successful that is has been translated into other languages [43]. Although at this point it would be premature to expect WeblabSim to reach the same level of popularity as OCW, this relatively minor change makes the client applet ready for this possibility.

3.3.5 Packaging and Delivery

The client program requires a level of access to the system beyond what is normally allowed to a Java applet loaded over the internet. By default, the Java security manager enforces constraints on what an applet can do because the applet is considered untrusted code. In general, applets loaded over the net are prevented from reading and writing files on the client file system, and from making network connections except to the originating host. In addition, applets loaded over the net are prevented from starting other programs on the client. Applets loaded over the net are also not allowed to load libraries, or to define native method calls. If an applet could define native method calls, that would give the applet direct access to the underlying computer.

In order to function, the WeblabSim client program needs to be able to bypass these restrictions. First, it needs to contact the service broker, which may be on a computer different from the one where the client applet originated.[5] Second, the client must be able to access the local file system in order to save simulation results.

To get around the limitations on untrusted applets, the WeblabSim client program is distributed as a digitally-signed Java archive (JAR) file. When the Java plug-in [44] runs an applet delivered in a signed JAR and loaded over the internet, it presents

[5]In fact, this is the case in the current setup. The client applet is loaded directly from the laboratory server, which is located on a machine that is different from the one hosting the service broker.

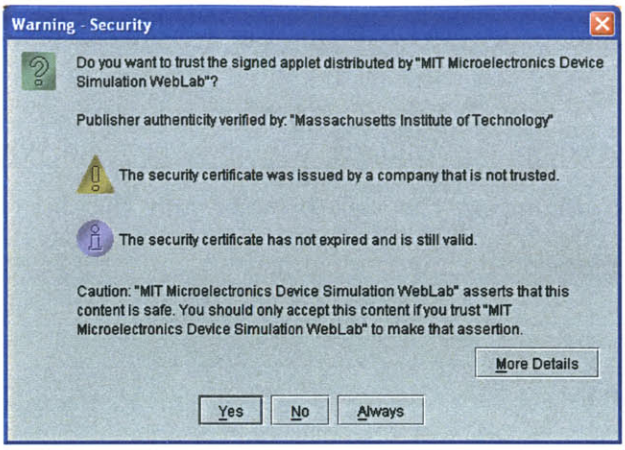


Figure 3-7: Security warning dialog displayed by the Java plug-in

the user with a warning dialog. The dialog explains who signed the applet and asks whether it should be trusted, as shown in Figure 3-7. The user responds by clicking one of the following buttons:

Yes The applet will be granted all permissions.

Deny The applet will be treated as an untrusted applet.

Always If selected, the applet will be granted all permissions. Any signed applet signed using the same certificate will be trusted automatically in the future, and no security dialog will pop up when this certificate is encountered again. This decision can be changed from the Java plug-in control panel.

More Info Users can examine the attributes of each certificate in the certificate chain in the JAR file.

We expect users to click either Yes or Always, so that the Java plug-in grants the client applet the permissions that it needs.

3.4 Resource Requirements

To run the applet correctly, the user must have the Java™2 Platform, Standard Edition version 1.4.2 or greater installed. The Java plug-in [44] component must

be present, and configured to work with the user's browser. For a list of supported platforms and browsers, see [45].

Informal testing on computers running Windows 2000 and Windows XP indicate that the client applet, including the web browser and the Java plug-in, consumes around 45–55 MB of memory during typical use.

3.5 Summary

In this chapter, we examined the WeblabSim client applet, which is WeblabSim's interface to its users. Before examining the WeblabSim applet itself, we briefly presented the program on which it was based, the WebLab client. We then described the user interface in detail, tying it back to the conceptual models developed in section 2.1. We also explained the design and implementation of the present WeblabSim client, focusing on the ways that it differs from its predecessor.

The next chapter describes the laboratory server, which runs the simulations that users create via the WeblabSim client.

Chapter 4

Laboratory Server Implementation

This chapter describes the implementation details of the laboratory server components. As discussed in Chapter 2, the laboratory server is responsible for maintaining the overall state of the online simulator, accepting simulations from the service broker, running these simulations, and reporting their results. To perform these tasks in a reliable, efficient and scalable manner, the server is divided into smaller and simpler components that work together.

The chapter begins with a brief review of laboratory server's internal structure and the communication pattern between the components. We then introduce Apache Axis, the web services platform on which the subsystems were implemented. The succeeding sections consider the divisions of the server, exploring each subsystem's design and implementation. Finally, we set the resource requirements for the laboratory server and present our testing methodology.

4.1 Laboratory Server Overview

The WeblabSim laboratory server is not one monolithic entity. Rather, it is composed of several interacting components, which together present the illusion of being a single laboratory server. There are three kinds of components: master servers (masters), worker servers (workers), and a data store. Although each one is potentially replicable, as explained in section 2.4.3, in this chapter we consider what we expect to be the

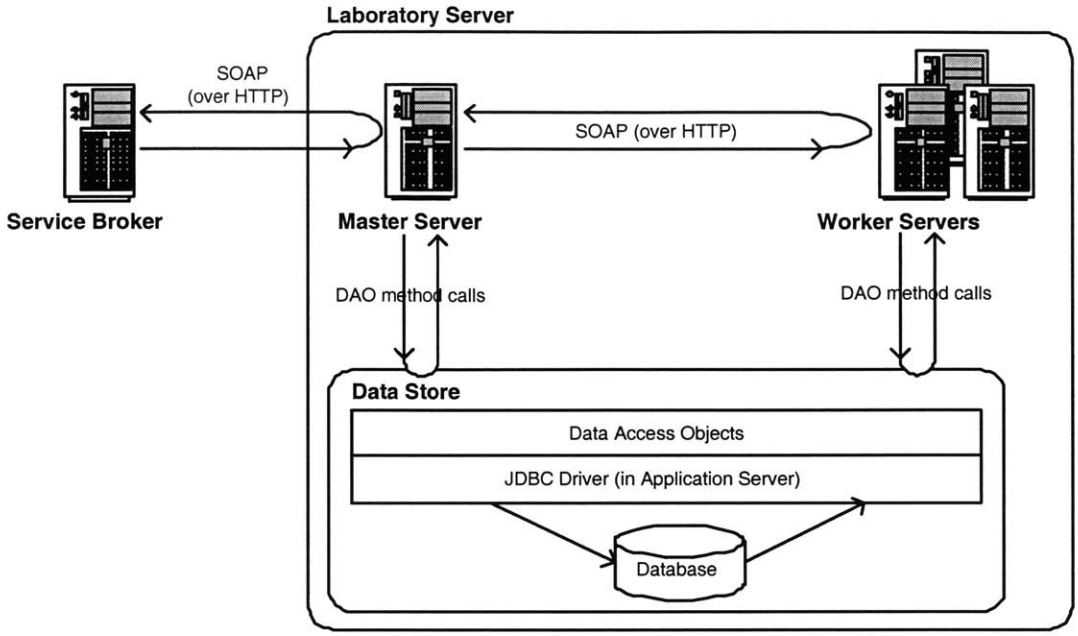


Figure 4-1: Typical laboratory server configuration, with one master server, one data store and several worker servers. Arrows in the figure indicate communication patterns between the server components.

typical configuration: one master server, one database, and a few worker servers.

The system enforces a strict isolation discipline between the components of the laboratory server. The master, workers and data store communicate only through well-defined channels, as illustrated in Figure 4-1.

- As the laboratory server's interface to the rest of the WeblabSim system, the master server acts as the server's controller. Based on the incoming request from the service broker, the master decides what needs to be done. It then either performs the task or delegates it to another part of the server. To delegate tasks, the master has to communicate with the rest of the server, but the master itself does not need to be contacted by the other components.

- The worker server exposes a SOAP web service interface to the master server through an HTTP endpoint [31]. The service has two methods: ping and submit. The ping method is a simple method that the master uses to check that a worker server exists at a given endpoint. The submit method is what actually runs a

simulation. The method expects a simulation specification in the body of the request message, and returns the simulation result in the body of the response.

- The master server and the worker servers depend on the data in the data store. In fact, every operation on both the master and the worker result in at least one database query. However, the master and workers do not access the database directly. Instead, they go through *data access objects* (DAOs) that mediate between application code and the data store [46]. The DAOs abstract and encapsulate the data store from the rest of the system. This makes it easier to switch database vendors, if necessary, or to implement performance optimizations like adding a cache.

Although the parts that comprise the laboratory server are logically separate, they do not have to be located on separate machines. They do not even have to be in different processes—the master and workers could be running in a single application server that has an embedded database (e.g., Berkeley DB [47], Cloudscape [48]). However, running the components on different machines/processes is highly recommended. The system is unlikely to benefit from replication if all the components are on the same computer, as they will simply be competing for the same limited resources. Also, using separate processes improves reliability by preventing unintended interactions between the components.

4.2 Data Store

The data store is used by the master and worker servers to store information about device models and device profiles. The master and the worker need these data to validate and run simulations. The store also maintains information about service brokers and usage groups, available resources and their associated permissions, and the status of simulation requests received by the master server.

Information about individual users are not kept in the laboratory server's data store. Instead, this information is managed by the service broker. The service broker

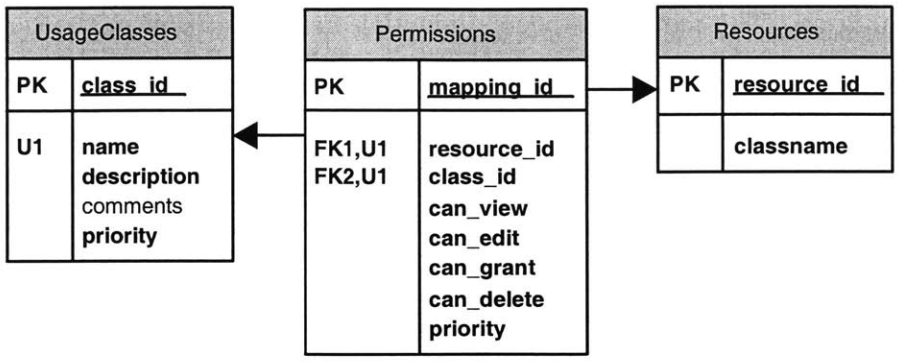


Figure 4-2: Tables storing security information

is also the repository for simulations performed by users, and their results. The specifications and results of simulations are kept in the data store temporarily, as part of the job record. This record is purged periodically.

4.2.1 Database Schema

Security

One of the goals in developing the laboratory server was to carefully separate the parts that were domain-dependent from those that could be applicable to other online simulators/laboratories. As security is generally a common concern, we carefully avoided references to domain-specific concepts like device models or device profiles in our security model (Figure 4-2).

In the model, the object of protection is the *resource*. A resource is just a number, the resource_id, that is associated with a Java class. The system expects that there are instances of the Java class stored in another table in the database, and that the table contains a column called resource_id. To determine the object that a given resource refers to, the system looks for the record with the same resource_id.

Permissions are associated with *usage classes*, which are the subjects of the permission. For a particular resource, all members of the same usage class have the same privileges. These can include the right to view and use the resource (can_view), edit its properties (can_edit), grant other usage classes permissions to the resource (can_grant), or delete the resource altogether (can_delete). The permission object

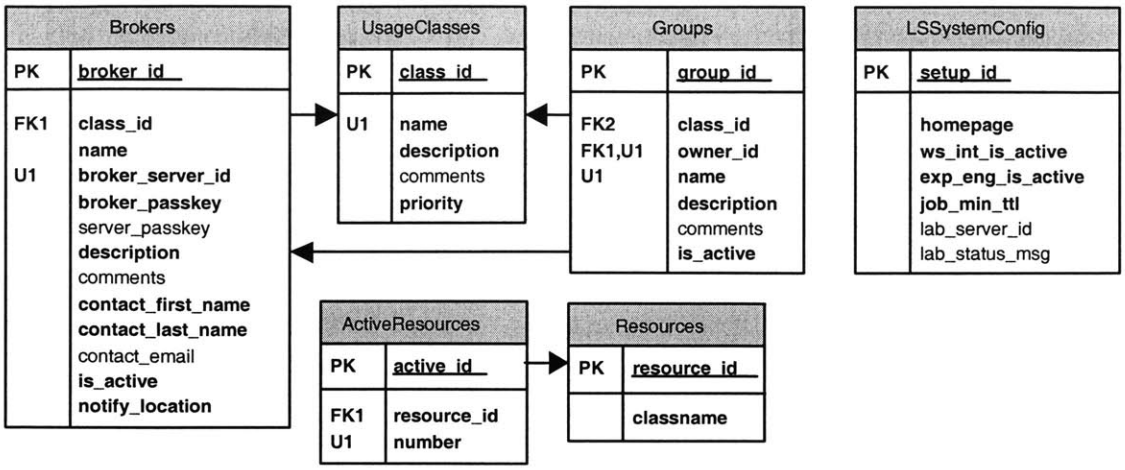


Figure 4-3: Tables storing the iLab laboratory server configuration

held by a usage class also determines the maximum priority of its simulations.

Service Brokers and User Groups

In the iLab Batched Experiment Architecture, the laboratory server delegates user-management functions to service brokers. Instead of storing information about all of its users, the laboratory server keeps track of the service brokers that it has associations with. Each service broker has a record in the Brokers table. Among other information about the broker, the record specifies the broker's default usage class.

The server also keeps track of groups of users as records in the Groups table. Groups are aggregations of users created by the service broker. For example, a service broker could have a group named "6012-students" to hold all the students taking the introductory microelectronics class at MIT. A group also belongs to a usage class. If the user's group is specified as a parameter to a laboratory server method, the class associated with the group is used to determine the user's permissions. Otherwise, the permissions of the broker's default usage class are used.

Devices

The tables in Figure 4-4 implement the data model described in section 2.1. The conceptual model was slightly modified, so that it could be represented efficiently

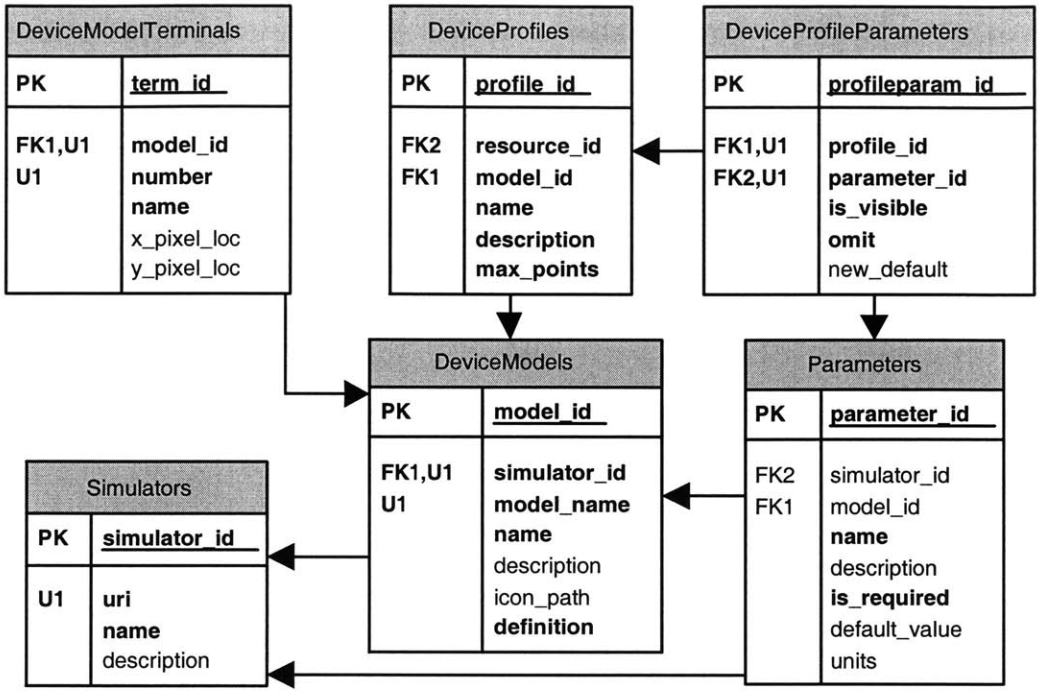


Figure 4-4: Tables storing information on simulators, device models and device profiles

as tables in a relational database. Table 4.1 summarizes how the concepts in the problem domain were represented as tables in the database.

Information about a device simulator (§2.1.3) is stored in the Simulators table. A device simulator has a unique Uniform Resource Identifier (URI) [49], which is how the WeblabSim system refers to it. For example, version 1.05.04 of the WinSpice simulator has the URI urn:simulator:WinSpice/1.05.04. For the benefit of administrators, a simulator also has a name and a description.

Database Table	Concepts
DeviceModels	device type, device model
DeviceModelTerminals	terminal
Parameters	model parameter, simulator parameter
Simulators	device simulator
DeviceProfiles	device profile
DeviceProfileParameters	model/simulator parameters, as modified by a device profile

Table 4.1: Mapping between database tables and concepts in the problem domain

The abstract notions of a device type (§2.1.1) and a device model (§2.1.2) have been unified in the database schema, and are represented as records in theDeviceModels table. There was not enough data common to different instances of a device type to warrant having separate tables for types and models. A device model is identified by its *qualified name*, a pair consisting of the URI of the simulator that implements the device model, and an arbitrary local part. The WeblabSim system uses qualified names to refer to devices, so each device must have a different qualified names. For instance, the independent voltage source in WinSpice version 1.05.04 has the local name V, so its qualified name is {urn:simulator:WinSpice/1.05.04}V. The record for a device model has other information: a name and description, a Uniform Resource Locator (URL) [50] pointing to a graphical representation of the device (icon_path), and a template that is used to create the input to the simulator (definition).

The DeviceModelTerminals table contains records describing the terminals of a device model. A terminal has a name and a number that indicates its ordinal position among the terminals of the device model. A pair of columns, x_pixel_loc and y_pixel_loc, indicate where the terminal is located in the device's image. This is used by the client applet when drawing the graphical representation of the simulation setup (as in Figure 3-2, for example).

Parameters are located in the Parameters table. A parameter must belong to either a simulator or a device model, never both. A parameter has a name, a description, and a field indicating whether or not the parameter must be specified to the simulator. Optionally, it can also have a default value and measurement units.

Device profiles (§2.1.4) exist as records in the DeviceProfiles table. A device profile holds a reference to the device model it is based on. It has a name, a description, and a limit on the number of datapoints that can be collected when it is used. It also has a resource ID that, as discussed earlier, serves as the profile's reference handle in the security subsystem. The profile's modifications to the model parameters are detailed in the DeviceProfileParameters table. A profile has a record in this table for every parameter in the device model, plus a record for every parameter of the associated device simulator. The entries indicate whether the parameter should be

hidden (is_visible), if it should be omitted from the underlying simulation (omit), and if provided, the new default value (new_default).

Active Resources

A table named ActiveResources determines which resources are currently active and accessible to users. The table has two data columns: resource_id, which names the active resource, and number, which determines the order in which the resources are listed. Note that the resource_id field does not have to be unique. This is so that a resource can appear multiple times in the laboratory configuration.

In WeblabSim, the resources are device profiles, so the values in the resource_id column refers to entries in the DeviceProfiles table.

Global Configuration Options

The laboratory server stores a number of system configuration options in the LSSystemConfig table. The data items are kept as columns of the table. There should only be one row in this table, the active row, and it should have a setup_id equal to 1. The options which are currently defined are listed in Table 4.2.

Job Processing

The laboratory server uses the JobRecord table to keep track of the simulationx specifications it has received, the simulations that it is running, and the results of any finished simulations. The master server maintains the entries in this table, filling in the columns as the job progresses from being queued, in progress, and then completed. Table 4.3 explains what the various fields of the job record mean.

The JobRecord table, and how it is used to implement a simulation queue, is discussed in more depth in section 4.4.2.

Column	Method	Description
server_id	none	used to identify the lab server when it calls Notify on the service broker
homepage	GetLabInfo	the page with more information about the online simulator
ws_int_is_active	all	indicates if the web service interface is active; if it is not, all web service calls result in a fault
min_ttl	GetExperimentStatus	the minimum guaranteed time before an experiment record is purged from the laboratory server
exp_eng_is_active	GetLabStatus	indicates if the server is currently accepting new simulations to process; if it is not, no progress is guaranteed on simulations that are submitted
status_msg	GetLabStatus	the current status of the laboratory server

Table 4.2: The columns of the LSSystemConfig table and their meanings. *Method* indicates which methods of the laboratory server web service interface is directly affected by the column.

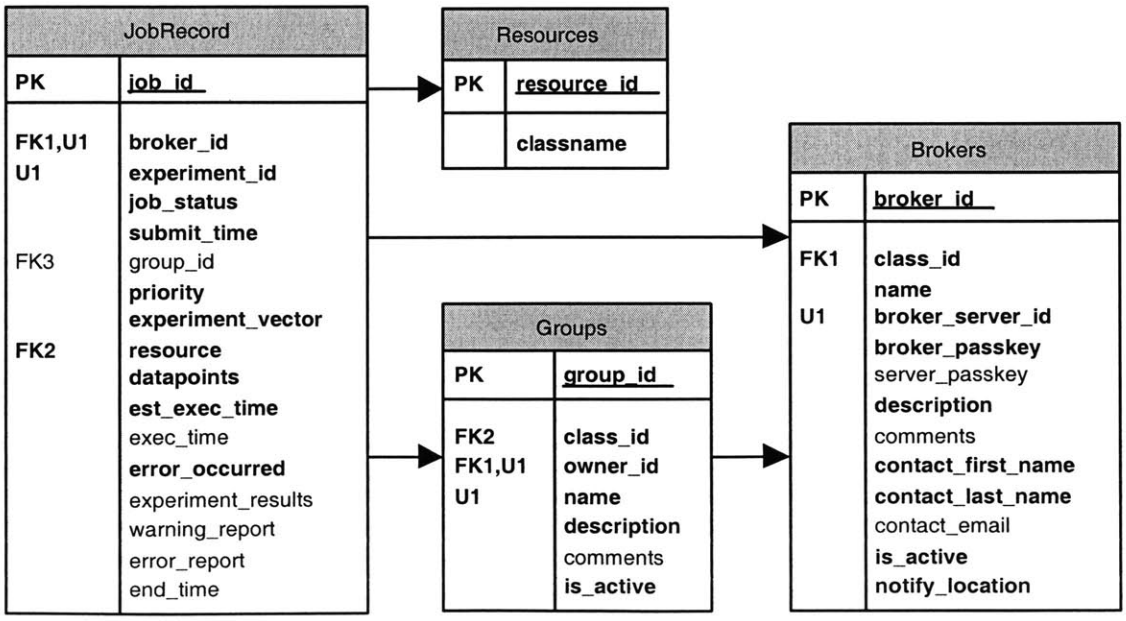


Figure 4-5: Tables storing information about jobs

Column	Set	Description
broker_id	q	service broker that submitted this job
experiment_id	q	experiment ID assigned to this job by the service broker that submitted it
job_status	q	indicates the current status of this job
group_id	q	effective group under which this job was submitted, or `NULL` if the job was not submitted under a group
priority	q	priority of this job
experiment_vector	q	simulation specification
resource	q	resource that is the subject of the simulation
datapoints	q	length of each result vector
est_exec_time	q	estimate of how long it will take to run this simulation
exec_time	s	when the simulation was started; that is, the time at which this simulation was sent to a worker server for execution
end_time	f	when the simulation ended
experiment_results	f	results of the simulation; this can be `NULL` if the simulation resulted in an error
warning_report	f	warning messages issued while running the simulation; this can be `NULL` if no warning messages were generated
error_report	f	error messages issued while running the simulation; this is `NULL` if the simulation finished successfully

Table 4.3: The columns of the JobRecord table and their meanings. *Set* indicates when the specified column is filled: when the job is inserted/enqueued (q), started running (s), or when it is finished (f).

4.2.2 Data Store Component Design

The data store component has three main divisions: the database server, an application server that manages connections to the database server, and a client library with the DAOs that are used by the master and worker servers. The three are arranged in a stack, with each layer operating at a higher level of abstraction than the one below it (see Figure 4-1).

Database Server

The database server is where the data are stored. WeblabSim assumes a relational database, such as Microsoft SQL Server™[51] or MySQL [52]. In a relational database, data are stored in the form of related tables. Relational databases make few assumptions on how the data are related, or on how they will be extracted. Having multiple views of the same data is useful when implementing a data model that exhibits varying levels of refinement, like that presented in section 2.1.

The precise wire protocol used to communicate with a database server depends on the database vendor and version. For example, Microsoft SQL Server™ uses the application-level protocol Tabular Data Stream (TDS) [53], whereas MySQL uses a completely different client/server protocol. WeblabSim solves these compatibility problems by hiding the database behind two layers of abstraction: an application server and a client library.

Application Server

The application server provides two important services to the data store component. First, it uses Java Database Connectivity (JDBC™) [54, 55] to provide access to a wide range of database servers. Thus, the application server effectively insulates the rest of the system from the specific details of the database server. From the point of view of layers higher up in the stack, the database is a generic relational database that can be accessed using Structured Query Language (SQL) [56] statements.

The application server also provides connection pooling and support for transac-

tions. Connection pooling is a technique used for sharing server resources among requesting clients. In terms of resources, database connections are relatively expensive to acquire, so applications benefit from having a set (a "pool") of cached connections already available. Although connection pooling is useful for sharing database resources optimally, it is purely an optimization technique.

On the other hand, transactions are extremely important in maintaining data consistency. A transaction is a set of statements that succeed or fail as a group (atomically). Even if the error that causes the transaction to abort occurs in the halfway through the execution of the transaction, the entire group of statements is rolled back, and the database is restored to its state before the transaction began. Transactions keep the data logically consistent. For example, adding a new device model could have the following steps:

1. Create a new device model with no parameters or terminals.
2. Create the model parameters, and add them to the device model.
3. Create the terminals, and add them to the device model.

Without transactions, if the operation failed at step 3, the database would have a device model with no terminals. This is explicitly forbidden by our data model (see Figure 2-4). With transactions, the same failure would cause the transaction to be rolled back, and it would be as if the attempt to add the device model had never happened.

Client Library

The preceding layers operate at the relational level, treating the data as if they were grouped into flat tables connected by relations. However, the master and worker servers are written in Java. Because Java is an object-oriented language, the fundamental unit of abstraction in a Java program is the *object*: an amalgamation of data and operations on data that is treated as a unified entity. Unlike the tables in a relational database, which are connected by one-way relations called keys, objects are naturally related by inheritance and composition [57].

WeblabSim employs Hibernate [58] to bridge the gap between the relational and object-oriented models. Hibernate is an object/relational mapping solution for Java environments. It maps a data representation from an object model to a relational data model with a SQL-based schema, and vice versa. Hibernate also provides data query and retrieval facilities. Through these facilities, we implemented the data access objects used by the master and worker servers to query the data store.

4.3 Worker Server

The WeblabSim worker server receives simulation specifications from the master server and runs them, returning the result back to the master. The worker server is built on top of Apache Axis [59], which provides the worker with a platform for building web services and a framework for modular message processing.

4.3.1 Apache Axis Web Services Framework

Apache Axis is an implementation of the Simple Object Access Protocol (SOAP) [30, 31]. SOAP is a lightweight protocol intended for exchanging structured information in a decentralized, distributed environment. SOAP is the backbone to web services, a new generation of cross-platform cross-language distributed computing applications. Axis is essentially a SOAP engine: a framework for constructing SOAP processors such as clients, servers, gateways, etc.

Brief Overview of SOAP

SOAP is a message-oriented protocol. Entities communicating through SOAP talk to each other by sending text messages, whose format is standardized by the SOAP specification. The primary part of this message has a MIME type of text/xml and contains the SOAP envelope. This envelope is an Extensible Markup Language (XML) [60] document. The envelope contains an optional header and a mandatory body. Attachments may be appended to the body.

```
<env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope">
 <env:Header>
  <n:alertcontrol xmlns:n="http://example.org/alertcontrol">
   <n:priority>1</n:priority>
   <n:expires>2001-06-22T14:00:00-05:00</n:expires>
  </n:alertcontrol>
 </env:Header>
 <env:Body>
  <m:alert xmlns:m="http://example.org/alert">
   <m:msg>Pick up Mary at school at 2pm</m:msg>
  </m:alert>
 </env:Body>
</env:Envelope>
```

Figure 4-6: SOAP message containing a SOAP header block and a SOAP body (from [30])

The body part of the envelope is always intended for the final recipient of the message, while the header entries may target the nodes that perform intermediate processing. SOAP provides a way for the client to specify which of the intermediate processing nodes has to deal with what header entry. Because headers are orthogonal to the main content of the SOAP message, they are useful in adding information to the message that does not affect the processing of the message body. Headers, for example, may be used to provide digital signatures for a request contained in the body. In this circumstance, an authentication or authorization server could process the header entry—independent of the body—-stripping out information to validate the signature.

The SOAP 1.2 specification [30] provides the sample envelope in Figure 4-6. In this example, an alert is being sent to a server somewhere on the web. The body contains the text of the actual alert. The header has instructions that control the parameters of the alert, in this case, its priority and when it expires. These pieces of information are relevant to the nodes that will deliver the alert, but not to the alert's final recipient.

Axis Architectural Overview [61]

The core task of Axis is processing SOAP messages (org.apache.axis.Message). When the central Axis processing logic runs, a series of handlers (org.apache.axis.Handler) are each invoked in order. The particular order is determined by how the Axis engine was configured. The object which is passed to each handler invocation is a message context (org.apache.axis.MessageContext). A message context is a structure which contains three important parts

1. a request message,
2. a response message, and
3. a bag of properties, stored as key-value pairs.

The Axis framework's job is simply to pass the message context through the configured set of handlers, each of which has an opportunity to do whatever it is designed to do with the message context.

The message path is shown in Figure 4-7. The small cylinders represent handlers, and the larger, enclosing cylinders represent chains (org.apache.axis.Chain). A chain is a handler consisting of a sequence of handlers which are invoked in turn.

A message arrives at a transport listener, which packages the protocol-specific data into a Message object and put the message into a MessageContext. The transport listener then passes it to the Axis engine.

The Axis engine first looks up the transport by name. The transport is an object which contains a request chain and/or a response chain. If a transport request chain exists, it will be invoked, thereby calling all the handlers specified in the transport request chain configuration. After the transport request chain, the engine locates a global request chain, if configured, and then invokes any handlers specified therein.

At some point during this processing, a handler must set the serviceHandler property of the message context. This field determines the handler executes service-specific functionality, such as making an remote procedure call on a back-end object. Services in Axis are typically instances of the org.apache.axis.handlers.soap.SOAPService class.

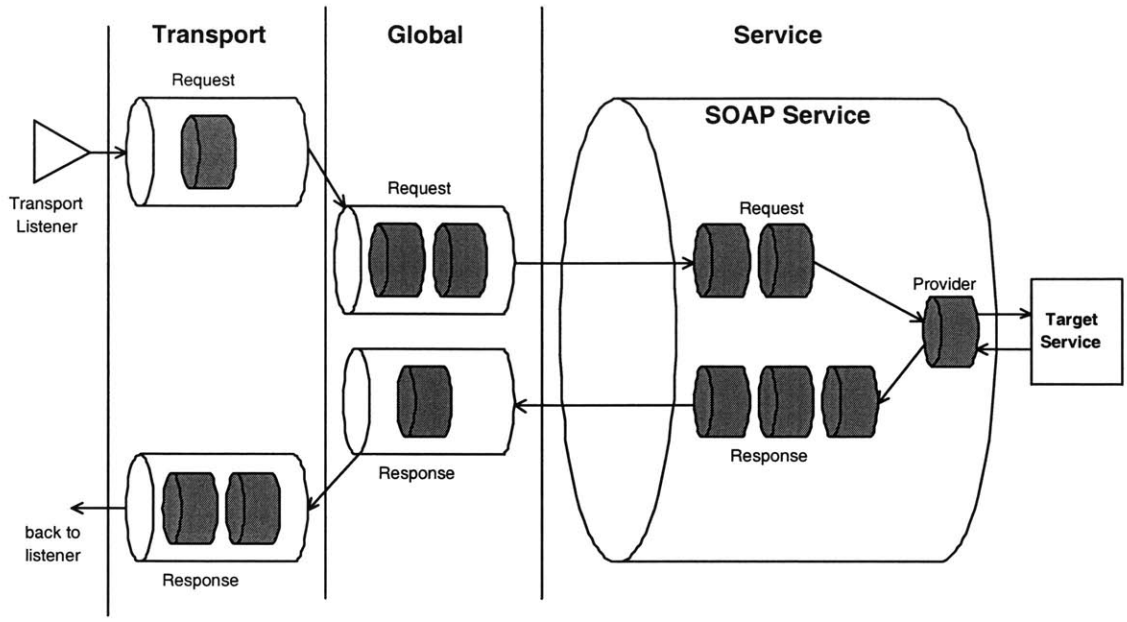


Figure 4-7: Path of a message through the Axis engine

The SOAP service may contain request and response chains similar to those at the transport and global levels. It must contain a provider, which is simply a handler responsible for implementing the actual back end logic of the service.

4.3.2 Worker Server Design

The worker server uses the Apache Axis system described in section 4.3.1 in two ways. First, Axis was used to implement the web service interface that the worker server exposes to the master server. Second, we took the handler framework used by the core Axis engine to process SOAP messages and on top of this, we implemented a generic handler framework for processing simulation specifications and results.

Web Service Interface

The web service interface is simple, consisting of only two methods. The ping method is a "null" method because invocations of this method return an empty reply. The master server uses ping to check that a given URL has an active worker server. The submit method takes a simulation specification in its body. If the simulation com-

Worker Server		Axis	
Concept	*Object*	*Concept*	*Object*
simulation specification	Experiment	request message	Message
simulation result	Result	response message	Message
device simulator	Simulator	SOAP service	SOAPService
handler that processes simulation specifications	ExperimentProcessor	handler on a request chain (transport/global/service)	Handler
handler that processes simulation results	ResultProcessor	handler on a response chain (transport/global/service)	Handler
handler that runs the device simulator	SimulatorProvider	handler that implements the actual back-end logic of the service	BasicProvider

Table 4.4: Objects in the simulation processing framework and the Axis concepts that they map to

pletes successfully, the results of the simulation are returned in the body of the reply message. Otherwise, the reply message contains a SOAP fault indicating the error that occurred.

Section E.1 in Appendix E describes the worker server's web service interface in more detail. The description therein is expressed in the Web Service Description Language (WSDL) [62], a language for describing web services and the possible ways to interact with them.

Simulation Processing Framework

The core of the worker server uses the Axis framework to process the simulation specification and the simulation results. The WeblabSim worker server framework consists of adapters that translate from WeblabSim concepts like "simulator", "simulation specification", "simulation results" to the analogous Axis concepts, which in this case would be "SOAP service", "request message" and "response message". Table 4.4 is a list of WeblabSim concepts and their Axis equivalents.

```
public interface ExperimentProcessor {
    public void invoke(Experiment theExperiment,
        MessageContext msgContext) throws Exception;
}
```

(a) Experiment processor interface

```
public interface Simulator {
    public Result invoke(Experiment theExperiment,
        MessageContext msgContext) throws Exception;
}
```

(b) Simulator interface

```
public interface ResultProcessor {
    public invoke(Result theResult, MessageContext msgContext,
        Experiment theExperiment) throws Exception;
}
```

(c) Result processor interface

Figure 4-8: Bridge interfaces between the simulation processing system and the Axis engine

Simulation specification The worker server embeds the simulation specification that it receives through the submit method in the body of a SOAP message. It creates a new message context and sets this message as the request message of the new context. When it is ready, the message context is sent to the Axis engine for processing via a special transport called weblabsim.

The simulation specification as it comes from the client is not well-suited to computerized processing. The specification was designed to correspond closely to the representation of the simulation in the WeblabSim applet. Although this ensures that the resulting specification is readily understandable by a human, it it is not organized in a way that makes it easy to manipulate. Thus, the very first thing that the weblabsim transport request chain does is to transform the specification into an Intermediate Input Format (IIF) (§E.2). An Axis handler that applies a stylesheet written in the XSL Transformations (XSLT) [63] language to the body of the SOAP message performs the conversion.

Specification processing The Axis engine acts on the message context as described in section 4.3.1. In theory, we could write plain Axis handlers that manipulate the text of the simulation specification directly. This soon gets rather tedious,

so we opted to represent the simulation specification as Java objects using the Java Architecture for XML Binding (JAXB) [64], and write handlers that act on the object representation directly.

Handlers that act on simulation specifications implement the ExperimentProcessor interface shown in Figure 4.3.2. The invoke method takes an Experiment instance, which is the object representation of an IIF document, and the message context. The message context is passed to invoke so that the method has access to the following: the raw text representation of the specification, the headers in the SOAP envelope, and the properties of the message context. The last part is especially important, because a handler on the request chain must set the serviceHandler property of the context, so that the Axis engine knows which simulator to run.

By itself, the Axis engine does not know how to call an ExperimentProcessor, much less where to get the instance of Experiment to pass to it. Because Axis works exclusively with handlers, the worker server wraps the experiment processor inside an Axis handler, edu.mit.weblabsim.axis.DomainHandler. DomainHandler takes three parameters: sync.in and sync.out, to be explained later, and domain.className, which is the fully-qualified class name of the experiment processor. When it is invoked, the handler does the following:

1. Retrieve the experiment object.
   (a) If sync.in is true, create and return a new experiment object from the body of the request message.
   (b) Otherwise, check the domain.experiment property of the message context. If this is a valid experiment object, return it.
   (c) Otherwise, create a new experiment object from the body of the request message.
2. Retrieve the experiment processor to delegate to.
   (a) Check for a cached instance of the experiment processor. If there is one, use it.

(b) Otherwise, create and use a new instance of the class specified by domain.className. Cache the object for future invocations.

3. Call the invoke of the experiment processor, passing in the experiment object.

4. Save the experiment object, which may have been modified.

   (a) Store the experiment object in the message context, under the domain.experiment key.

   (b) If sync.out is true, convert the experiment object to its XML representation and replace the body of the request message with the new simulation specification.

Simulations In the simulation processing system, device simulators are represented by instances of the Simulator interface, shown in Figure 4.3.2. The interface hides the details of how the simulator is run behind the invoke method, whose signature is quite simple: it takes an experiment and the message context, and returns a result. The actual implementation of the interface is likely to be more complicated; the class for the WinSpice 1.05.04 simulator has seven helper methods and two configuration options.

In the Axis system, device simulators are represented by services. To the Axis engine, a simulator is just another SOAP service, with a request chain, a response chain, and a provider that does the actual back-end logic. The request chain contains handlers that process the simulation in a way that is specific to the particular device simulator. The provider is a thin wrapper around the Simulator object that instantiates it, passes it any options, and delegates to it all method calls. The response chain has handlers that account for particular idiosyncrasies of the simulator, and also handlers that undo or reverse the changes made by the handlers in service request chain. Axis identifies services by their namespace, which is a URI. The namespace of a simulator service is the same as URI assigned to the device simulator that it runs.

The data from running the simulation is expressed in the Intermediate Output Format (IOF) (§E.3), which is a way to represent multidimensional values using XML.

Aside from the individual data points, the IOF structures keep track of the value's name, dimensions and optionally, its units. The body of the SOAP response message holds the results. Also, the message context holds an object representation of the results (an instance of the Result class) under the domain.result key.

Result processing The worker server depends on the Axis engine to invoke the handlers that process the results of the simulation. The Axis engine invokes a sequence of handlers, starting with those in the service response chain, and ending with the ones in the weblabsim transport response chain.

Just like the handlers that process the specification in the request chain, the handlers that process the result are typically written to work with Experiment and Result objects directly. These handlers implement the ResultProcessor interface shown in Figure 4.3.2. Apart from the result object and the message context, the invoke method of the result processor also takes an experiment object as one of its parameters. Having the experiment object around is very useful, because it is often necessary to examine the input when processing the results. For example, the handler the calculates user-defined functions must be able to access the functions' definitions, which is in the experiment specification.

The same DomainHandler that adapts experiment processors so that the Axis engine can use them also works for result processors.

Simulation result When the message context comes back after being processed by the Axis engine, the worker server's web service interface expects the result of the simulation to be in the body of the response message. This result is returned, unchanged, to the master server. The master expects the simulations results to be in a format that is different from that used internally by the worker server (IOF). Therefore, last thing the weblabsim transport does is convert the result object to its XML representation, and then transform the resulting text to the format expected by the master.

4.3.3 Sample Configuration: WinSpice 1.05.04

This section describes how we configured a WeblabSim worker server to target the WinSpice 1.05.04 circuit simulator. The chains and handlers below are specified as Web Service Deployment Descriptors (WSDD) [65, 66] located in a file named server-config.wsdd. Unless otherwise specified, the class implementing the handler (specified in brackets after the name) is either an experiment processor or a result processor, and therefore has to be wrapped in a DomainHandler.

Transport Request Chain (weblabsim)

The transport request chain does the processing that will have to be performed, regardless of which simulator is targeted by the specification.

WeblabIIFTransformer [edu.mit.weblabsim.worker.handlers.XSLTransformer]

This handler translates the simulation specification from the format submitted by the client applet into the IIF. XSLTransformer works with the text in the SOAP message, so it is a true Axis handler.

DeviceResolver [edu.mit.weblabsim.worker.handlers.DeviceResolver]

The specification submitted by the client applet contains only the resource ID of the device. The WeblabIIFTransformer, being a mechanical XSLT stylesheet, cannot do the lookups required to create a complete simulation specification in IIF. Instead of using the qualified name of the device and the simulator's URI, WeblabIIFTransformer simply uses a placeholder, and leaves it to the DeviceResolver to fill in the details. These details include:

1. Converting the placeholder for the device name into the qualified model name of the device profile corresponding to the given resource ID.
2. Replacing the placeholder simulator URI with the URI of the simulator associated with the device model.
3. If the profile specified any hidden device or simulator parameters, adding the hidden parameters to the specification with their default values.

4. If the profile specified any omitted device or simulator parameters, making sure that the parameters are not present in the specification.

SimulatorMapper [edu.mit.weblabsim.worker.handlers.SimulatorMapper]

This handler sets the serviceHandler property of the message context to the correct simulator service. It does this by looking for an Axis service whose namespace matches the simulator URI in the specification.

Simulator Service (WinSpice 1.05.04)

The service request chain contain handlers that perform processing that is specific to simulator that it targets. For example, if a simulator cannot even sweep the value of a voltage source, a handler could emulate a sweep by making multiple copies of the circuit, one for each voltage value in the sweep. The WinSpice simulator service does not have any handlers in its request chain.

Axis services are configured by specifying a provider, which is a handler that does the back-end logic for the service. The behavior of the provider may be customized through parameters. As mentioned above, the service must be given a namespace that corresponds to the URI of the simulator that it is targeting. For the WinSpice service, the provider is edu.mit.weblabsim.axis.SimulatorProvider, a thin wrapper around an instance of the Simulator interface.

WinSpice_1.05.04 [edu.mit.weblabsim.simulators.winspice.WinSpiceRunner]

This implementation of Simulator creates a SPICE3 input from the simulation specification and hands it to the WinSpice executable. The output of the simulation is redirected to a file, which it parses into multidimensional values. These values are returned in a Result object.

This service is configured by two parameters. executable points to the location of the WinSpice executable in the local filesystem. The code running the worker server must have the proper permissions to read and execute the file. debug is an option that can turn on "debug mode". In debug mode, the handler does not

delete the WinSpice input and output files after it finishes; the administrator can then examine the files to help diagnose problems.

The service request chain contain handlers to process results in a way that is specific to simulator that just ran. These handlers are deployed to account for idiosyncrasies of the simulator, or to undo the changes made by the handlers in the service request chain. Continuing our previous example, the primitive simulator above probably needs a handler in the response chain to put together the data from the multiple circuits created by the request chain.

CurrentFix [edu.mit.weblabsim.simulators.winspice.FixCurrentDirectionHandler]

WinSpice has one handler in the service response chain. This handler fixes a mismatch in the sign conventions of WeblabSim and WinSpice. In WebLab, currents are measured such that current flowing into a terminal is positive. The way we implemented the simulation in WinSpice, currents flowing into a terminal are negative. This handler takes all the currents in the result and negates the signs of the datapoints.

Transport Response Chain (weblabsim)

The transport response chain consists of handlers that must be invoked regardless of which simulator produced the result.

Calculator [edu.mit.weblabsim.worker.handlers.CalculatorHandler]

The Calculator handler parses the user-defined functions in the input into expressions. It determines the order in which to evaluate them (§A.4) and then runs the calculator on each expression. The resulting values are placed back in the Result object.

DownloadFilter [edu.mit.weblabsim.worker.handlers.DownloadFilter]

This handler filters the values in the result, keeping only those that the user marked for download.

Temperature [edu.mit.weblabsim.worker.handlers.TemperatureOutputHandler]

The results expected by the WeblabSim applet includes the temperature at which the simulation was performed. The Temperature handler extracts this information from the simulation specification and places it in the results as another variable.

ResultSyncOut [edu.mit.weblabsim.worker.handlers.BasicResultProcessor]

So far, all the handlers in the response chain have been acting on the Result object without updating the body of the SOAP response message. The system deliberately allows this skew, because it is prohibitively expensive to serialize the Result object into the SOAP message after every handler. However, the final conversion to the format expected by the WeblabSim applet acts on the body of the SOAP response, so when that handler is invoked, the two must be in synchrony. This handler ensures that by marshaling the result object into text and putting that in the body of the response.

WeblabOIFTransformer [edu.mit.weblabsim.worker.handlers.XSLTransformer]

This handler translates the simulation results from the OIF into the format expected by the client applet. Like WeblabIIFTransformer, this handler uses XSLTransformer, which is a true Axis handler.

4.4 Master Server

The WeblabSim master server implements the iLab laboratory server web service interface (§B.1). The methods in the interface can be divided into two broad classes: those that return information about the laboratory (GetLabInfo, GetLabStatus, GetLabConfiguration, GetEffectiveQueueLength), and those involved in submitting experiments and retrieving their results (Validate, Submit, Cancel, GetExperimentStatus, RetrieveResults).

While implementing these methods, we balanced two sometimes-competing goals. First, the master server should be relatively fast and small. In our vision of the

typical laboratory server, one master server would take care of communicating with the service broker and delegate the computing-intensive tasks to the (multiple) worker servers. But because all messages must go through the master server, it will eventually become the bottleneck in the system. Minimizing the amount of processing required in the master server increases the load level at which the laboratory server is overloaded.

At the same time, we strove to keep the master server sufficiently general. While the master server was under development, the actual implementation of the API between the service broker and the laboratory server was prone to change, particularly in the areas of authentication and authorization. A simple, extensible master server mitigated the adverse effect of these changes by making it easy to add new functions or modify existing ones.

Also, our experience with the MIT Microelectronics WebLab laboratory server[1] and subsequent attempts to adapt it to other experimental domains indicated that a monolithic design has several shortcomings. Gerardo Viedma took the WebLab laboratory server code and modified it to implement a laboratory server for the 6.302 Feedback Systems Web Laboratory [67]. In his report [68], he observed that:

- The existing laboratory server is tightly coupled to the MIT Microelectronics WebLab. Thus, it is hard to identify all the places in the source code that need to be updated when porting the server to an new laboratory domain. (pp. 17–18, 20)

- Dependencies between laboratory server modules make it difficult to change parts of the server in isolation, because "most of the components [have] to be working (or removed) for the others to also work or be tested." (p. 19)

Eventually, we decided that it would be easier to adopt a new design for the master server instead of teasing out and replacing the domain-dependent parts of the WebLab laboratory server. This made it easier for us to fulfil our goals of speed and

[1]The WebLab laboratory server was developed, along with the rest of the Batched Experiment Architecture, by the iLab group (see [22]). James Hardison was the lead developer for the laboratory server.

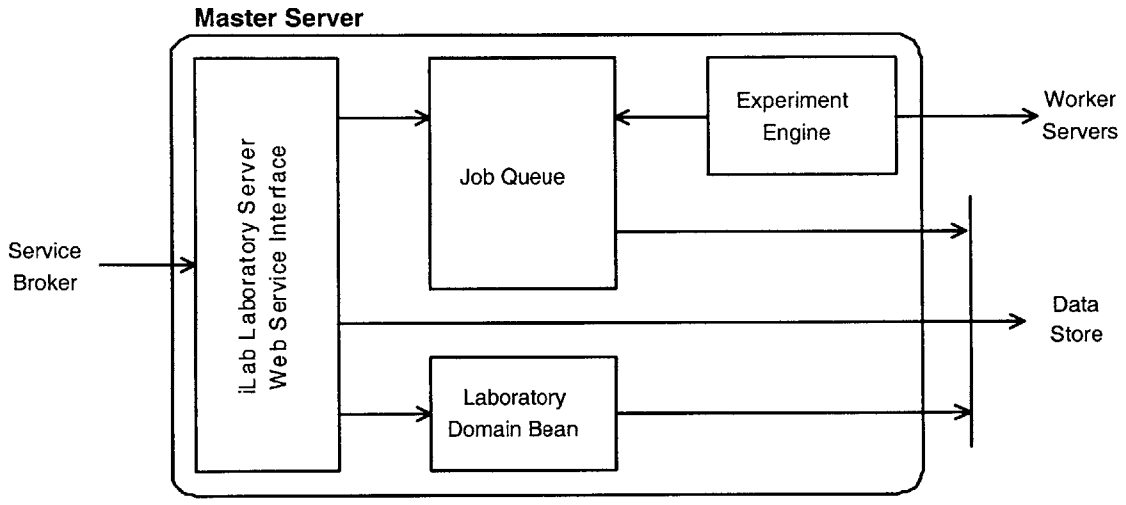


Figure 4-9: Parts of the master server, showing dependencies among them and with other WeblabSim components

generality. Although the internal structures of the master server and the WebLab laboratory server are rather different, the logic behind the master borrowed heavily from WebLab's code. This is especially true in the web service interface with the service broker, as the WebLab laboratory server was the de facto reference implementation of this interface.

4.4.1 Master Server Design

The WeblabSim master server is designed around three modules, as shown in Figure 4-9. Of these modules, the job queue and the experiment engine are intended to be generic and directly usable in many laboratory servers; only the laboratory domain bean contains logic that is specific to WeblabSim. The master server web service integrates the three components.

The master server works as follows. The service broker calls methods on the web service interface. The method calls are processed by a series of handlers that enforce authentication, authorization and access policies at the service broker and user group level. The web service then dispatches the request to the appropriate component. Generally, the job queue handles submissions, cancellations and queries about the length of the queue or the status of a simulation. The laboratory domain bean

validates simulation specifications and returns the laboratory configuration. Methods asking about the general state of the laboratory server are referred to information in the data store.

At certain times, the system causes the experiment engine to process simulation specifications in the job queue. The experiment engine receives the simulations, selects a worker server to run the simulation, and then sends it to the worker for it to run. When the results of the simulation come back, the experiment engine updates the job record and notifies the service broker that the simulation is complete. How the experiment engine is triggered, how it selects which job to process, and how it chooses which worker server to choose are all policy decisions that depend on the implementation of the master server components.

The job queue, experiment engine, and laboratory domain are Enterprise JavaBean (EJB) [69] components. The EJB architecture is a component architecture for the development of component-based distributed applications. The proper use of EJBs helps makes applications scalable, transaction and multi-user secure—certainly properties that the master server should have. In addition, EJBs provide excellent support for reconfiguring applications right when they are deployed. An EJB has an interface, the "bean interface", through which they are accessed. Different implementations of this interface can be substituted for each other when the application is deployed. For the master server, this means that the server administrator can try out different implementations of the job queue, experiment engine and laboratory domain until he finds a combination that works well for him.

4.4.2 Job Queue

The job queue bean (JobQueue) implements a priority queue of jobs. A priority queue is a data structure for maintaining a set of elements, each with an associated value called a key. The priority queue has operations for inserting an element into the queue, finding the element in the queue with the largest key, and extracting the element in the queue with the largest key [70]. Effectively, the keys determine the order of the jobs in the queue.

The job queue bean interface has the following methods:

- int[] getQueueLength(int priority)

  Returns the current length of the queue, from the point of view of a new job to be inserted at the given priority. The length of the queue is measured according to two metrics. The first, returned at index zero, is the number of jobs in the priority queue that are ahead of the hypothetical new job. The second, returned at index one, is an estimate of the number of milliseconds before the new job finishes.

- int[] getQueueLength(JobRecord job)

  Returns the current length of the queue, from the point of view of the given job. For a description of the return value, see getQueueLength(int).

- List getSnapshot()

  Returns a snapshot of the current state of the priority queue. The snapshot has all the queued jobs, arranged in the order that they will be extracted.

- int[] submit(JobRecord job)

  Submits a new job to the queue.

- JobRecord getJob(int broker, int experimentID)

  Retrieves the job record with the given experiment ID which came from the the specified service broker.

- boolean cancel(JobRecord job)

  Cancels the specified job.

- JobRecord select()

  Returns the job at the head of the priority queue, and marks it as currently in progress. This method is used by the experiment engine to get the next simulation to run.

- void finish(JobRecord job)

  Marks the given job as finished. The information in the data store is updated

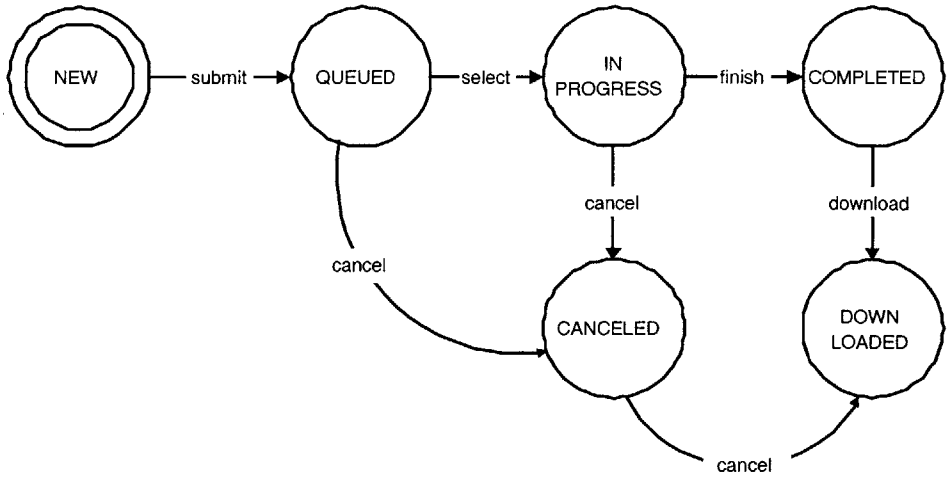


Figure 4-10: Life cycle state diagram of a job record

with the information in the job record object.

- JobRecord download(int broker, int experimentID)

  Like getJob, returns the job record with the given experiment ID, and which came from the specified service broker. In addition, the job is marked as having been downloaded already.

A job is represented by a JobRecord object, which is the object form of an entry in the JobRecord table (Figure 4-5). A job record is in one of the following states, as recorded by its jobStatus property: NEW, QUEUED, IN PROGRESS, CANCELED, COMPLETED and DOWNLOADED. The methods of the job queue cause the job record to transition between the states, as shown in Figure 4-10.

The order of queued jobs is determined by an SQL query provided to the job queue when it is configured. Initially, we ordered the jobs by priority, breaking ties by favoring jobs what were submitted earlier. For example, if there were three queued jobs, J_1 and J_3 submitted at 12:00:00 with priorities 10 and 0, respectively, and J_2 submitted at 12:01:00 with priority 10, then the queue would yield the jobs in the following order: J_1, J_2, and then J_3. However, it turns out that this simple ordering does not guarantee liveness. Because a job always overtakes a job with lower priority regardless of when they were submitted, a job with sufficiently low priority can be

kept on the queue indefinitely if higher-priority jobs keep arriving. To prevent this from happening, the current implementation arranges the jobs by the minute in which they were submitted. Ties are broken by favoring higher-priority jobs. For instance, had we used this strategy in the previous example, the order would be J_1, J_3, J_2. Clearly, under the new ordering jobs cannot be starved indefinitely, as a job can only be overtaken by others submitted in the same minute.

4.4.3 Experiment Engine

The experiment engine bean (ExperimentEngine) takes jobs from the queue and dispatches them to a worker server for execution. The experiment engine interface has one method, int runExperiments(WorkerService server, int limit) that sends up to limit number of simulations to the given worker server. The implementation of this method is not particularly interesting: it simply takes a job from the queue, sends the corresponding simulation to the worker server, then awaits the results. The interesting policy decisions are in the surrounding code, which determines when to call the method (when should the system check the job queue for pending simulations?), and what arguments arguments to pass to it (which worker server should execute a given simulation?).

Polling Trigger

The experiment engine in the current implementation of WeblabSim uses a polling mechanism (PollingTriggerBean) to determine when the check the queue and which worker server to use. Each worker server is associated with a timer that goes off periodically. Whenever it fires, a timer event occurs. The handler for the timer event invokes the runExperiments method, passing the worker server associated with the timer.

In this scheme, the frequency with which the system checks the queue is bounded by the poll interval. If the interval is set to 30 seconds, then the system will look at the queue every 30 seconds (approximately), although sometimes it may check the queue

more frequently.[2] Which worker server is used is determined by the order in which the application server delivers the timer events. The precise order is nondeterministic, but over time it should average out to an even distribution of load between the available worker servers.

4.4.4 Laboratory Domain

The laboratory domain bean (LabDomain) contains all the domain-dependent code in the master server. Its interface has the following methods:

- ValidationReport validate(JobRecord job)

  Validates the experiment specification in the given job record. The validation rules are domain-dependent; for WeblabSim, the experiment (simulation) specification is described in section D.2 of Appendix D. If the specification is valid, the method sets the resource and numDatapoints properties of the job record.

- String getLabConfiguration(List resources)

  Returns the laboratory configuration, given a list of resources available to the caller. The format of the laboratory configuration is domain-dependent; the format used by WeblabSim is described in section D.1 of Appendix D.

4.5 Resource Requirements

The server components of WeblabSim are packaged as Java enterprise applications, and need to be run inside an application server that supports the Java™2 Platform, Enterprise Edition (J2EE™) 1.4 [71]. The reference implementation, Sun Java™System Application Server Platform Edition 8 Update 1 (SJSASPE8) [72] was used while developing the application, and is expected to suffice for deployment. SJSASPE8 requires a minimum of 256 MB of memory (512 MB recommended), with at least 250 MB free space on the hard drive (500 MB recommended). The release

[2] This happens in runExperiments, which checks the queue immediately after running a simulation until it reaches the number of simulations to which it was limited, or it finds that the queue is empty.

notes [73] report that the following platforms are supported: Sun Solaris 8 and 9, RedHat Enterprise Linux 2.1 and 3.0, Microsoft Windows 2000 Professional, Server and Advanced Server (SP4+) and Windows XP Professional (SP1+).

The laboratory server requires a connection to a database, where it stores its configuration. It relies on the application server to manage the database connections automatically. Hence, the database server chosen and its driver must be compatible with the application server.[3]

The server depends on a simulator, an external program, to run the simulations. The resource requirements of different simulators vary. A simulator cannot be used with WeblabSim unless the platform running the laboratory meets the resource requirements of the simulator.

During development, we verified that the server operates on a machine with the following specifications:

- 4 2.0 GHz Intel Xeon™processors
- 1.5 GB of RAM
- Windows 2000 Server (SP4)
- Sun Java™System Application Server Platform Edition 8 Update 1
- Microsoft SQL Server™2000 (SP3)

In this configuration, a minimal laboratory server consisting of one master server (§4.4) and one worker server (§4.3) has a memory footprint of approximately 100 MB in its quiescent state (no requests being serviced). We expect memory usage to stay under 512 MB during typical use.

4.6 Testing

Our testing strategy consisted of continuous unit testing during development, load tests on the laboratory server, and informal acceptance testing of the entire system.

[3]For SJSASPE8, a list of compatible drivers is included with the release notes [73].

The following sections describe our methodology for unit testing and load testing in further detail. Acceptance tests, which are tests conducted to determine whether or not a system has its planned functionality, were done informally by using the WeblabSim client to define simulations, run them, and graph their results.

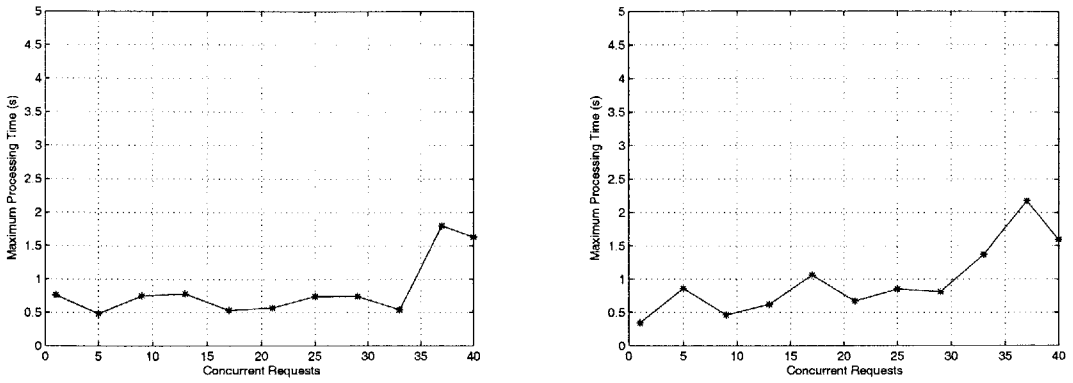
4.6.1 Unit Tests

A unit test is a collection of tests designed to verify the behavior of a single unit within the program. While implementing the laboratory server, we regularly wrote unit tests for the part of the system that we were developing. WeblabSim's testbed consists of 57 unit tests containing 242 test cases. (There are actually more than 242 individual tests, because each test case often contains multiple tests.) The tests cover all parts of the laboratory server. We ran these unit tests often, and did not commit any code to the repository until it had passed all of them.

4.6.2 Load Tests

Load testing generally refers to the practice of modeling the expected usage of a software program by simulating multiple users accessing the program's services concurrently. We conducted load tests on the laboratory server components. One master server and one worker server were deployed on the machine described in section 4.5 above. The master and worker were each barraged with anywhere from 1–40 concurrent requests to a web service method. The method chosen was the one that required the most server resources to process. On the worker, it was a call to Submit with a simulation that generated a total of 402 values. On the master, the test script called GetLabConfiguration, which returned a laboratory configuration containing three device profiles.

The results of these tests revealed that the master server and the worker server can handle up to at least 40 concurrent requests, with a worst-case latency of less than 2.5 seconds (Figure 4-11) to service each one. For comparison, the design of the



(a) Worker Server (Submit) (b) Master Server (GetLabConfiguration)

Figure 4-11: Results of stress tests performed on the master server and the worker server.

WebLab laboratory server limits it to running only one job at a time,[4] with a mean time of 8.61 seconds per job.[5] Data collected by the iLab team in the spring of 2004 indicate that the service broker (deployed as part of the WebLab system) performs well even with a user load of up to 26 users per hour [22]. Thus, this stress test suggest that WeblabSim will be just as responsive as WebLab, if not more so.

4.7 Summary

In this chapter, we described the internals of the WeblabSim laboratory server. We gave an overview of the three components that comprise the laboratory server and the interfaces between them. We then discussed in turn the data store, the worker servers and the master server, describing in detail how we designed and implemented each component. We then estimated the system resources required by a laboratory server during normal operation. We concluded the chapter by presenting our testing methodology and some of our results.

[4]Because WebLab actually runs the experiments submitted to it, all jobs must go through the semiconductor parameter analyzer. In the current WebLab design, there is only one analyzer, which forces serial execution.

[5]Based on data collected from January 1, 2004 through August 15, 2004 (min = 0.28 s, max = 100.06 s, mean = 8.61 s, median = 6.41 s, std. deviation = 8.24 s).

Chapter 5

Conclusion

We have designed and implemented the MIT Device Simulation WebLab ("Weblab-Sim"), an online simulator for exploring the behavior of microelectronic devices. We have deployed the WinSpice [1] circuit simulator on the WeblabSim system, and we have used it successfully to run simulations of several kinds of microelectronic devices.

Furthermore, we believe that WeblabSim satisfies the design values of extensibility, efficiency and reliability that we outlined in Chapter 2. Our system was designed to be modular, making WeblabSim easy to extend and improve. Guided by the iLab Batched Experiment Architecture, we divided the system into three parts that communicate through web service interfaces: the laboratory server, service broker and laboratory client. The reduced level of coupling allowed us to take advantage of existing components that provided the functionality that we desired (e.g., user management, experiment storage, authentication and authorization). This also lets us take advantage of improvements in each of the components without changing the rest of the system substantially (if at all).

Whenever possible, we designed the parts that we implemented ourselves with an eye for extensibility. This led us to make design decisions like splitting the logic of the laboratory server into master and worker servers (§§4.4–4.3), implementing the web service interfaces on the Apache Axis platform (§4.3.1), and processing simulations through a system of configurable handlers (§4.3.2). These features allow WeblabSim to adapt relatively easily to changes in operating conditions and system requirements.

We have developed a system that uses its resources efficiently. This is evidenced by the modest system requirements for the client applet and the laboratory server, as well as the system's solid performance during load testing. The extensive testbed that we used while developing WeblabSim, coupled with the results of our load tests, makes us confident that we have designed and implemented a reliable system.

Although the present version of WeblabSim is feature-complete and functioning reliably and efficiently, there is still room for improvement. In particular, we are working on the following projects.

Management interface WeblabSim should have a management interface where laboratory administrators can install, view, edit and delete device models, device simulators and device profiles. This is the system's most pressing need. Presently, these tasks can only be done by editing the database directly. Certain functions, like deploying a new simulator, even requires re-compiling and re-installing the WeblabSim system. Having to do these manually makes them error-prone, which in turn directly impacts the system's availability and reliability. To eliminate these problems, we are currently developing a web-based management interface very much like that used by WebLab [74].

Monitoring interface In order to manage a system effectively, administrators must be able to carefully monitor the state of the system. We currently accomplish this by pervasive logging throughout the system. However, parsing log files is tedious work, and we think that the data they contain can be presented in a more informative way. We are also working on a web-based monitoring interface, which we eventually hope to integrate with the management system described above.

Integration with WebLab As mentioned previously, our ultimate goal is to integrate WebLab and WeblabSim into a single online laboratory. A student can then perform both measurements and simulations, and compare their results. We believe that the conjunction of an online remote laboratory and an online simulator will be

a valuable resource in microelectronics education.

At this point, WeblabSim's core features are complete, and it is ready to be deployed for use in a classroom setting. We plan to use WeblabSim in Fall 2004 as part of the MIT graduate-level class Integrated Microelectronic Devices (6.720). During this time, we hope to evaluate the usability of the system and the utility of using an online device simulator in microelectronics teaching.

Appendix A

Expression Language for User-Defined Functions

A.1 Values

There are two types of values in the calculator: scalars and multidimensional values. A value of scalar type (also referred to as scalar value, or simply, a scalar) denotes a single double-precision floating-point number, as specified by Java [75].[1] However, positive and negative infinities, and the special Not-A-Number (NaN) value are not valid scalars. Multidimensional values denote an ordered collection of numbers. Each number in the collection is called an element.

Types are represented by sequence of integers enclosed in angle brackets. The ith number in a type T, denoted by T_i, indicates the length of the type along dimension i. The total number of elements in a value of type T with n dimensions is given by $size[T] = \prod_1^n T_i$. For example, `<>` is the scalar type, which has one element; `<5>` is the type of a vector with five elements; and `<2,3>` is the type of a 2×3 matrix, which has 6 elements.

The elements of a value of type T are assigned integers from 0 through $size[T] - 1$. These correspond to the element's index in the value's row-major representation. For

[1] In its definition of floating-point values, the *Java Language Specification* references the *IEEE Standard for Binary Floating-Point Arithmetic* [76].

instance, the array $\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$ has the row-major representation $[1, 2, 3, 4, 5, 6]$, so the number 3 is at index 2. The notation $v[i]$ refers to the ith element of the value v.

A.1.1 Operations, Value Normalization

Unless otherwise noted, all mathematical operations on two scalars are defined as in [75]. The referenced definitions sometimes specify positive infinity, negative infinity and NaN as the result of certain operations. These special values are not valid values in the expression language. Therefore, when these values occur, they are automatically normalized as follows:

- The largest double-precision floating-point value is substituted in place of positive infinity and NaN.
- The arithmetic negation of the largest double-precision floating-point value is substituted in place of negative infinity.

A.1.2 Type Compatibility, Value Promotion

In general, it is an error for an operator to act on operands of different types. However, if one operand is a scalar, and the other is a multidimensional value, the scalar is promoted to a multidimensional value of the same type by replicating the value of the scalar.

For example, the following operations result in an error:

- $[1, 2, 3] + [1, 2, 3, 4]$ (two vectors of different lengths)
- $[1, 2] + \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$ (a vector and a matrix)

However, it is legal to operate on a scalar and a multidimensional value: $1 + [1, 2, 3] = [1, 1, 1] + [1. 2, 3] = [2, 3, 4]$.

A.2 Lexical Structure

This section specifies the lexical structure of the expression language. Expressions are written in ASCII. The characters forming the definitions of the user-defined functions are reduced to a sequence of input elements, which are white space, literals, identifiers, separators and operators of the syntactic grammar.

A.2.1 White Space

White space is defined as the ASCII space, horizontal tab, form feed, carriage return and line feed characters. The language ignores all white space characters.

A.2.2 Literals

A literal is the representation of a floating-point value in the input. A literal can be a plain number, a number written in scientific notation, a number coupled with a metric scale, or a symbolic literal. The metric scales are f (10^{-15}), p (10^{-12}), n (10^{-9}), u (10^{-6}), m (10^{-3}), k (10^{3}), M (10^{6}) and G (10^{9}). There are three symbolic constants: the electronic charge "q" (1.602177×10^{-19}), Boltzmann's constant "k" (1.380658×10^{-23}), and the permittivity of free space "e" (8.854188×10^{-12}).

```
Literal:
    BareLiteral
    SciNotationLiteral
    MetricNotationLiteral
    SymbolicLiteral

BareLiteral:
    Digits "." [Digits]
    "." Digits
    Digits

SciNotationLiteral:
    BareLiteral [ExponentPart]

MetricNotationLiteral:
    BareLiteral [ScaleSuffix]
```

```
SymbolicLiteral:
    "k" | "q" | "e"

ExponentPart:
    ("e" | "E") ["+" | "-"] Digits

ScaleSuffix:
    "f" | "p" | "n" | "u" | "m" | "k" |"M" | "G"

Digit:
    Digit
    Digits Digit

Digit:
    "0"–"9"
```

A.2.3 Identifiers

An identifier is an unlimited-length sequence of letters and digits, the first of which must be a letter. A letter consists of uppercase and lowercase alphabetic characters, underscore and the dollar sign. A digit is a numeral from zero through nine.

```
Identifier:
    IdentifierChars (except SymbolicLiteral)

IdentifierChars:
    Letter (Letter | Digit)*

Letter:
    "a"–"z" | "A"–"Z" | "_" | "$"
```

A.2.4 Separators

The following three ASCII characters are the separators (punctuators): left parenthesis (`(`), right parenthesis (`)`), and the semicolon (`;`).

A.2.5 Operators

The language has six operators, formed from ASCII characters: `+`, `-`, `*`, `/`, `%` and `^`.

A.3 Expressions

The evaluation of an expression denotes a value. This section specifies the meanings of expressions and the rules for their evaluation.

A.3.1 Type

An expression has a type that is known at the time the expression is submitted as part of a simulation. The rules for determining the type of an expression are explained separately below for each kind of expression. Before an expression is evaluated, the system checks that its type is valid. If the expression does not pass this type-checking phase, it must be rejected. Otherwise, evaluation must complete without any system errors.

A.3.2 Evaluation Order

Because the language guarantees that an expression with a valid type can be evaluated successfully, there is no need to impose a particular evaluation order within an expression. Implementations are free to take advantage of algebraic identities such as the associative law to rewrite expressions into a more convenient computational order.

A.3.3 Primary Expressions

Primary expressions include most of the simplest kinds of expressions, from which all others are constructed: literals, variables and function invocations. A parenthesized expression is also treated syntactically as a primary expression.

```
Primary:
    Literal
    Variable
    FunctionInvocation
    "(" Expression ")"
```

Literals

A literal (§A.2.2) denotes a fixed, unchanging value. The type of a literal is always the scalar type (<>).

Variables

A variable is an identifier that refers to a value. The value may be a result of the simulation (i.e., the current or voltage at a terminal, or a model observable) or the result of another user-defined function. If the variable refers to a result of the simulation, its type is the type of the value that is referenced. If it refers to another user-defined function, the variable has the same type as the expression that defines the function.

It is important to note that the type of a variable can be determined at the time the expression is submitted by examining the simulation specification. For example, if the specification calls for a VAR1 sweep from 0.0 V to 1.0 V in 10 mV increments, it follows that all simulation results will be vectors with 101 elements. Thus, all variables that refer to these results are of type <101>.

Let $V(e)$ is the set of variables used in the expression e, and let $v \in V(e)$ such that v refers to a user-defined function. Then we say that v is a dependency of e, or that e depends on v.

A.3.4 Function Invocation Expressions

A function invocation expression is used to invoke a built-in function. Table A.1 lists the predefined functions and their meanings.

Function	Return Value
ABS(*expr*)	the absolute value of *expr*
AT(e_1, e_2)	returns the element of e_1 at index e_2
AVG(*expr*)	the average of all the elements in *expr*
COND(e_1, e_2, e_3, e_4)	if $e_1 < e_2$, returns e_3, otherwise, returns e_4
DELTA(*expr*)	the difference between successive elements of *expr*
DIFF(e_1, e_2)	the derivative of e_1 with respect to e_2, which is equivalent to DELTA(e_1)/DELTA(e_2)
EXP(*expr*)	the base of natural logarithms (e) raised to the power of *expr* (§A.3.6)
INTEG(e_1, e_2)	the numerical approximation to the integral of e_1 as a function of e_2
LGT(*expr*)	the base 10 logarithm of ABS(*expr*)
LOG(*expr*)	the natural logarithm of ABS(*expr*)
MAVG(e_1, e_2)	the moving average of the elements in e_1, taken e_2 elements at a time
MAX(*expr*)	the largest element in *expr*
MIN(*expr*)	the smallest element in *expr*
SQRT(*expr*)	the square root of each element in ABS(*expr*)

Table A.1: Functions built in to the expression language

```
FunctionInvocation:
    BuiltinFunctionName "(" [ArgumentList] ")"

BuiltinFunctionName:
    "ABS" | "AT" | "AVG" | "COND" | "DELTA" | "DIFF" | "EXP"
    "INTEG" | "LGT" | "LOG" | "MAVG" | "MAX" | "MIN" | "SQRT"

ArgumentList:
    Expression
    ArgumentList "," Expression
```

A.3.5 Unary Operators

The unary operators are + and -. Expressions with unary operators group right-to-left, so that +-x means +(-x).

The type of a unary plus or minus expression is the type of its operand. The value of a unary plus expression is the value of its operand. The value of a unary minus expression is the arithmetic negation of the value of its operand.

```
UnaryExpression:
    Primary
    "-" UnaryExpression
    "+" UnaryExpression
```

A.3.6 Exponentiation Operator

The exponentiation operator is `^`. Exponentiation expressions are right-associative (they group right-to-left), so that `x^y^z` means `x^(y^z)`. The type of an exponentiation expression is the promoted type of its operands.

```
PowerExpression:
    UnaryExpression
    UnaryExpression "^" PowerExpression
```

When evaluating an exponentiation expression like `x^y`, the values of both operands are promoted to the type of the expression, resulting in two values of the same type. If both operands are scalars, then the result is simply the left operand raised to the power of the right operand, after value normalization (§A.1.1). Otherwise, element at index i of the result is obtained by evaluating `x[i]^y[i]`.

A.3.7 Multiplicative Operators

The operators `*`, `/`, and `%` are called the multiplicative operators. They have the same precedence and are syntactically left-associative (they group left-to-right). The type of a multiplicative expression is the promoted type of its operands.

```
MultiplicativeExpression:
    PowerExpression
    MultiplicativeExpression "*" PowerExpression
    MultiplicativeExpression "/" PowerExpression
    MultiplicativeExpression "%" PowerExpression
```

When evaluating an multiplicative expression, the values of both operands are promoted to the type of the expression, resulting in two values of the same type. If both operands are scalars, then the binary operator is applied to the values of the operands, as defined in [75]. The result is then normalized (§A.1.1). Otherwise, the result is the multidimensional value obtained by applying the binary operator

pairwise to each element of the operands (see §A.3.6).

A.3.8 Additive Operators

The operators + and - are called the additive operators. They have the same precedence and are syntactically left-associative (they group left-to-right). The type of a multiplicative expression is the promoted type of its operands.

```
AdditiveExpression:
    MultiplicativeExpression
    AdditiveExpression "+" MultiplicativeExpression
    AdditiveExpression "-" MultiplicativeExpression
```

When evaluating an multiplicative expression, the values of both operands are promoted to the type of the expression, resulting in two values of the same type. If both operands are scalars, then the binary operator is applied to the values of the operands, as defined in [75]. Otherwise, the result is the multidimensional value obtained by applying the binary operator pairwise to each element of the operands (see §A.3.6).

A.3.9 Expression

An expression is any additive expression.

```
Expression:
    AdditiveExpression
```

A.4 Evaluation Order of User-Defined Functions

When a simulation contains more than one user-defined function, the language constrains the order in which they are evaluated. In general, the dependencies (§A.3.3) of a function f must be evaluated before the function f itself.

Given two functions i and j, if j is a dependency of i, then j must be evaluated before i. If i is a dependency of j, then i must be evaluated before j. Otherwise, i and j may be evaluated in any order.

Appendix B

iLab Batched Experiment Architecture APIs

B.1 Service Broker to Laboratory Server API

The following listing was taken from a document written by Judson Harward, last modified on October 29, 2003 [77].

B.1.1 Data Types

```
public class LabStatus {
    /* true iff lab is accepting experiments */
    public boolean online;

    /* domain-dependent human-readable text describing status of lab
       server */
    public String labStatusMessage;
}

public class WaitEstimate {
    /* number of experiments currently in the execution queue that would
       run before the hypothetical new experiment */
    public int effectiveQueueLength;

    /* [OPTIONAL, < 0 if not supported] estimated wait (in seconds) until
       the hypothetical new experiment would begin, based on the other
       experiments currently in the execution queue */
    public double estWait;
}
```

```
public class ValidationReport {
    /* true iff the experiment specification would be (is) accepted
       for execution. */
    public boolean accepted;

    /* domain-dependent human-readable text containing non-fatal warnings
       about the experiment. */
    public String[] validationWarningMessages;

    /* [if accepted == false] domain-dependent human-readable text
       describing why the experiment specification would not be (is not)
       accepted. */
    public String validationErrorMessage;

    /* [OPTIONAL, < 0 if not supported] estimated runtime (in seconds) of
       this experiment. */
    public double estRuntime;
}

public class SubmissionReport {
    /* see Validate() */
    public ValidationReport vReport;

    /* guaranteed minimum time (in hours, starting now) before this
       experimentID and associated data will be purged from the lab
       server */
    public double minTimeToLive;

    /* see GetEffectiveQueueLength() */
    public WaitEstimate wait;
}

public class LabExperimentStatus {
    public ExperimentStatus statusReport;

    /* guaranteed minimum remaining time (in hours) before this
       labExperimentID and associated data will be purged from the lab
       server */
    public double minTimeToLive;
}

public class ExperimentStatus {
    /* Indicates the status of this experiment.  1 iff waiting in the
       execution queue, 2 iff currently running, 3 iff terminated normally,
       4 iff terminated with errors (this includes cancellation by user in
       mid-execution), 5 iff cancelled by user before execution had begun,
       6 iff unknown labExperimentID */
    public int statusCode;

    /* see GetEffectiveQueueLength() */
    public WaitEstimate wait;

    /* [OPTIONAL, <0 if not used] estimated runtime (in seconds) of this
```

```
        experiment */
    public double estRuntime;

    /* [OPTIONAL, <0 if not used] estimated remaining runtime (in seconds)
       of this experiment, if the experiment is currently running */
    public double estRemainingRuntime;
}

public class ResultReport {
    /* Indicates the status of this experiment.  1 iff waiting in the
       execution queue, 2 iff currently running, 3 iff terminated normally,
       4 iff terminated with errors (this includes cancellation by user in
       mid-execution), 5 iff cancelled by user before execution had begun,
       6 iff unknown labExperimentID */
    public int statusCode;

    /* [REQUIRED if experimentStatus == 3, OPTIONAL if experimentStatus == 4]
       an opaque, domain-dependent set of experiment results */
    public String experimentResult;

    /* opaque description of the lab configuration for the execution of this
       experiment */
    public String labConfiguration;

    /* [OPTIONAL, null if unused] a transparent XML string that helps to
       identify this experiment.  Used for indexing and querying in generic
       components which can't understand the opaque experimentSpecification
       and experimentResults */
    public String xmlResultExtension;

    /* [OPTIONAL, null if unused] a transparent XML string that helps to
       identify any blobs saved as part of this experiment's results.*/
    public String xmlBlobExtension;

    /* domain-dependent human-readable text containing non-fatal warnings
       about the experiment including runtime warnings*/
    public String[] executionWarningMessages;

    /* [REQUIRED if experimentStatus == 4] domain-dependent human-readable
       text describing why the experiment terminated abnormally including
       runtime errors */
    public String executionErrorMessage;
}
```

B.1.2 Interface Methods

```
/**
 * Checks on the status of the lab server.
 *
 * @return the status of the lab server
 */
public LabStatus GetLabStatus();

/**
 * Checks on the effective queue length of the lab server. The notion of an
 * "effective queue" is the answer to the following question:
 * hypothetically, if a user belonging to the specified
 * userGroup were to submit a new experiment right now with the
 * specified priorityHint, how many of the experiments
 * currently in the execution queue would run before the new experiment?
 *
 * @param userGroup effective group of the user submitting the hypothetical
 *     new experiment
 * @param priorityHint indicates a requested priority for the hypothetical
 *     new experiment. Possible values range from 20 (highest priority) to
 *     -20 (lowest priority); 0 is normal. Priority hints may or may not be
 *     considered by the lab server.
 * @return the effective queue length of the lab server
 */
public WaitEstimate GetEffectiveQueueLength(String userGroup,
        int priorityHint);

/**
 * Gets general information about a lab server.
 *
 * @return  a URL to a lab-specific information resource, e.g. a lab
 * information page
 */
public String GetLabInfo();

/**
 * Gets the lab configuration of a lab server.
 *
 * @param userGroup effective group of the user requesting the lab
 * configuration
 * @return an opaque, domain-dependent lab configuration
 */
public String GetLabConfiguration(String userGroup);

/**
 * Checks whether an experiment specification would be accepted if
 * submitted for execution.
 *
 * @param experimentSpecification an opaque, domain-dependent experiment
 *     specification
 * @return effective group of the user submitting this experiment
 */
```

```
public ValidationReport Validate(String experimentSpecification);

/**
 * Submits an experiment specification to the lab server for execution.
 *
 * @param experimentID the identifying token that can be used to inquire
 *     about the status of this experiment and to retrieve the results
 *     when ready
 * @param experimentSpecification an opaque, domain-dependent experiment
 *     specification
 * @param userGroup effective group of the user submitting this experiment
 * @param priorityHint indicates a requested priority for this experiment.
 *     Possible values range from 20 (highest priority) to -20
 *     (lowest priority); 0 is normal. Priority hints may or may not be
 *     considered by the lab server.
 * @return a submission report
 */
public SubmissionReport Submit(int experimentID,
        String experimentSpecification, String userGroup, int priorityHint);

/**
 * Cancels a previously submitted experiment. If the experiment is already
 * running, makes best efforts to abort execution, but there is no guarantee
 * that the experiment will not run to completion.
 *
 * @param experimentID a token that identifies the experiment
 * @return true iff the experiment was successfully removed
 *     from the queue (before execution had begun). If false, the user
 *     may want to call GetExperimentStatus for more detailed
 *     information.
 */
public boolean Cancel(int experimentID);

/**
 * Checks on the status of a previously submitted experiment.
 *
 * @param experimentID a token that identifies the experiment
 * @return the status of the experiment
 */
public LabExperimentStatus GetExperimentStatus(int experimentID);

/**
 * Retrieves the results from (or errors generated by) a previously
 * submitted experiment.
 *
 * @param experimentID a token that identifies the experiment
 * @return a report describing the result
 */
public ResultReport RetrieveResult(int experimentID);
```

B.2 Client to Service Broker API

The following listing was taken from a document written by David Zych, and last modified on September 16, 2003 by Judson Harward [78].

B.2.1 Data Types

The API between the client and the service broker also uses the data types defined in the API between the service broker and the laboratory server, in addition to the following structure.

```
public class ClientSubmissionReport {
    /* see Validate() */
    public ValidationReport  vReport;

    /* a token that can be used to inquire about the status of this
       experiment and to retrieve the results when ready. */
    public int experimentID;

    /* guaranteed minimum time (in hours, starting now) before this
       experimentID and associated data will be purged from the lab
       server */
    public double minTimeToLive;

    /* see GetEffectiveQueueLength() */
    public WaitEstimate wait;
}
```

B.2.2 Interface Methods

```
/**
 * Checks on the status of the lab server.
 *
 * @param labServerID which lab server to use
 * @return the status of the lab server
 */
public LabStatus GetLabStatus(String labServerID);

/**
 * Checks on the effective queue length of the lab server. The notion of an
 * "effective queue" is the answer to the following question:
 * hypothetically, if this user were to submit a new experiment right now
 * with the specified priorityHint, how many of the experiments
 * currently in the execution queue would run before the new experiment?
```

```
 *
 * @param labServerID which lab server to use
 * @param priorityHint indicates a requested priority for the hypothetical
 *     new experiment. Possible values range from 20 (highest priority) to
 *     -20 (lowest priority); 0 is normal. Priority hints may or may not be
 *     considered by the lab server.
 * @return the effective queue length of the lab server
 */
public WaitEstimate GetEffectiveQueueLength(String labServerID,
        int priorityHint);

/**
 * Gets general information about a lab server.
 *
 * @param labServerID which lab server to use
 * @return  a URL to a lab-specific information resource, e.g. a lab
 * information page
 */
public String GetLabInfo(String labServerID);

/**
 * Gets the lab configuration of a lab server.
 *
 * @param userGroup effective group of the user requesting the lab
 * configuration
 * @return an opaque, domain-dependent lab configuration
 */
public String GetLabConfiguration(String userGroup);

/**
 * Checks whether an experiment specification would be accepted if
 * submitted for execution.
 *
 * @param labServerID which lab server to use
 * @param experimentSpecification an opaque, domain-dependent experiment
 *     specification
 * @return effective group of the user submitting this experiment
 */
public ValidationReport Validate(String labServerID,
        String experimentSpecification);

/**
 * Submits an experiment specification to the lab server for execution.
 *
 * @param labServerID which lab server to use
 * @param experimentSpecification an opaque, domain-dependent experiment
 *     specification
 * @param priorityHint indicates a requested priority for this experiment.
 *     Possible values range from 20 (highest priority) to -20
 *     (lowest priority); 0 is normal. Priority hints may or may not be
 *     considered by the lab server.
 * @param emailNotification if true, the service broker will make one
       attempt to notify the user (by email to the address with which the
       user's account on the service broker is registered) when this
```

```
       experiment terminates
 * @return a client submission report
 */
public SubmissionReport Submit(String labServerID,
        String experimentSpecification, int priorityHint,
        boolean emailNotification);

/**
 * Cancels a previously submitted experiment. If the experiment is already
 * running, makes best efforts to abort execution, but there is no guarantee
 * that the experiment will not run to completion.
 *
 * @param experimentID a token that identifies the experiment
 * @return true iff the experiment was successfully removed
 *     from the queue (before execution had begun). If false, the user
 *     may want to call GetExperimentStatus for more detailed
 *     information.
 */
public boolean Cancel(int experimentID);

/**
 * Checks on the status of a previously submitted experiment.
 *
 * @param experimentID a token that identifies the experiment
 * @return the status of the experiment
 */
public LabExperimentStatus GetExperimentStatus(int experimentID);

/**
 * Retrieves the results from (or errors generated by) a previously
 * submitted experiment.
 *
 * @param experimentID a token that identifies the experiment
 * @return a report describing the result
 */
public ResultReport RetrieveResult(int experimentID);

/**
 * Sets an client item value in the user's opaque data store.
 *
 * @param name name of the client item to set
 * @param value new value of the client item
 */
public void SaveClientItem(String name, String value);

/**
 * Returns the value of a client item in the user's opaque data store.
 *
 * @param name name of the client item to get
 * @return value current value of the client item
 */
public String GetClientItem(String name);

/**
```

```
 * Removes a client item from the user's opaque data store.
 *
 * @param name name of the client item to remove
 */
public void RemoveClientItem(String name);

/**
 * Enumerates the names of all client items in the user's opaque data
 * store.
 *
 * @return array containing the names of all client items in the user's
 *      opaque data store
 */
public String[] ListClientItems();

/**
 * Adds a user-specified annotation to a previously submitted experiment.
 *
 * @param experimentID a receipt returned from a previous call to Submit
 * @param annotation a human-readable comment to help the user identify
 *      this experiment again in the future
 */
public void Annotate(int experimentID, String annotation);
```

Appendix C

WebLab Client Document Formats

The following sections define the documents used by the WebLab client. For each document type, we give its Document Type Definition (DTD) [60] and an example of a document that conforms to the DTD.

C.1 Laboratory Configuration

C.1.1 Document Type Definition

[1]

```
<!ELEMENT labConfiguration (device*)>
<!ATTLIST labConfiguration
    lab          CDATA  #FIXED   "MIT Microelectronics Weblab"
    specversion  CDATA  #REQUIRED>
<!ELEMENT device (name, description, imageURL, terminal+, maxDataPoints)>
<!ATTLIST device
    id    CDATA  #REQUIRED
    type  CDATA  #REQUIRED>
<!ELEMENT name (#PCDATA)>
<!ELEMENT description (#PCDATA)>
<!ELEMENT imageURL (#PCDATA)>
<!ELEMENT terminal (label, pixelLocation, maxVoltage, maxCurrent)>
<!ATTLIST terminal
    portType    (SMU | VSU | VMU)  #REQUIRED
    portNumber  CDATA              #REQUIRED>
```

[1] The syntax and semantics of a Document Type Definition (DTD) are defined in the XML specification [60].

```
<!ELEMENT label (#PCDATA)>
<!ELEMENT pixelLocation (x, y)>
<!ELEMENT x (#PCDATA)>
<!ELEMENT y (#PCDATA)>
<!ELEMENT maxVoltage (#PCDATA)>
<!ELEMENT maxCurrent (#PCDATA)>
<!ELEMENT maxDataPoints (#PCDATA)>
```

C.1.2 Sample Document

```
<?xml version="1.0" encoding="utf-8" standalone="no" ?>
<!DOCTYPE labConfiguration
    SYSTEM "http://weblab2.mit.edu/xml/labConfiguration.dtd">
<labConfiguration lab="MIT Microelectronics Weblab" specversion="0.1">
    <device id="5" type="nMOSFET">
        <name>6.012 nMOSFET</name>
        <description>Aren't nMOSFETs just swell?</description>
        <imageURL>http://weblab.mit.edu/Version5.0/icons/NMOS.gif</imageURL>
        <terminal portType="SMU" portNumber="2">
            <label>Gate</label>
            <pixelLocation>
                <x>121</x>
                <y>94</y>
            </pixelLocation>
            <maxVoltage>2.0</maxVoltage>
            <maxCurrent>0.1</maxCurrent>
        </terminal>
        <terminal portType="SMU" portNumber="3">
            <label>Source</label>
            <pixelLocation>
                <x>195</x>
                <y>156</y>
            </pixelLocation>
            <maxVoltage>5.0</maxVoltage>
            <maxCurrent>0.1</maxCurrent>
        </terminal>
        <terminal portType="SMU" portNumber="4">
            <label>Bulk</label>
            <pixelLocation>
                <x>235</x>
                <y>94</y>
            </pixelLocation>
            <maxVoltage>5.0</maxVoltage>
            <maxCurrent>0.1</maxCurrent>
        </terminal>
        <terminal portType="SMU" portNumber="1">
            <label>Drain</label>
            <pixelLocation>
                <x>195</x>
                <y>23</y>
            </pixelLocation>
            <maxVoltage>5.0</maxVoltage>
            <maxCurrent>0.1</maxCurrent>
        </terminal>
        <maxDataPoints>500</maxDataPoints>
    </device>
</labConfiguration>
```

C.2 Experiment Specification

C.2.1 Document Type Definition

```
<!ELEMENT experimentSpecification
    (deviceID, terminal+, userDefinedFunction*)>
<!ATTLIST experimentSpecification
    lab          CDATA  #FIXED   "MIT Microelectronics Weblab"
    specversion  CDATA  #REQUIRED>
<!ELEMENT deviceID (#PCDATA)>
<!ELEMENT terminal (vname, ((iname, mode, function, compliance?) |
                            (function, compliance?) |
                            (mode))?)>
<!ATTLIST terminal
    portType   (SMU | VSU | VMU) #REQUIRED
    portNumber CDATA             #REQUIRED>
<!ELEMENT vname (#PCDATA)>
<!ATTLIST vname download (true | false) #REQUIRED>
<!ELEMENT iname (#PCDATA)>
<!ATTLIST iname download (true | false) #REQUIRED>
<!ELEMENT mode (#PCDATA)>
<!ELEMENT function ((scale?, start, stop, step) |
                    (offset, ratio) |
                    (value))>
<!ATTLIST function
    type (VAR1 | VAR2 | VAR1P | CONS) #REQUIRED>
<!ELEMENT scale  (#PCDATA)>
<!ELEMENT start  (#PCDATA)>
<!ELEMENT stop   (#PCDATA)>
<!ELEMENT step   (#PCDATA)>
<!ELEMENT offset (#PCDATA)>
<!ELEMENT ratio  (#PCDATA)>
<!ELEMENT value  (#PCDATA)>
<!ELEMENT compliance (#PCDATA)>
<!ELEMENT userDefinedFunction (name, units, body)>
<!ELEMENT name  (#PCDATA)>
<!ATTLIST name
    download (true | false) #REQUIRED>
<!ELEMENT units (#PCDATA)>
<!ELEMENT body  (#PCDATA)>
```

C.2.2 Sample Document

```
<?xml version="1.0" encoding="utf-8" standalone="no" ?>
<!DOCTYPE experimentSpecification
    SYSTEM "http://weblab2.mit.edu/xml/experimentSpecification.dtd">
<experimentSpecification lab="MIT Microelectronics Weblab" specversion="0.1">
    <deviceID>1</deviceID>
    <terminal portType="SMU" portNumber="1">
```

```
        <vname download="true">VDS</vname>
        <iname download="true">IDS</iname>
        <mode>V</mode>
        <function type="VAR2">
            <scale>LIN</scale>
            <start>0.05</start>
            <stop>0.5</stop>
            <step>0.1</step>
        </function>
        <compliance>0.1</compliance>
    </terminal>
    <terminal portType="SMU" portNumber="2">
        <vname download="true">VG</vname>
        <iname download="true">IG</iname>
        <mode>V</mode>
        <function type="VAR1">
            <scale>LIN</scale>
            <start>0.0</start>
            <stop>4.5</stop>
            <step>0.05</step>
        </function>
        <compliance>0.1</compliance>
    </terminal>
    <terminal portType="SMU" portNumber="3">
        <vname download="true">VGRD</vname>
        <iname download="true">IGRD</iname>
        <mode>V</mode>
        <function type="CONS">
            <value>0.0</value>
        </function>
        <compliance>0.1</compliance>
    </terminal>
    <userDefinedFunction>
        <name download="true">SQRTID</name>
        <units>A</units>
        <body>SQRT(IDS)</body>
    </userDefinedFunction>
</experimentSpecification>
```

C.3 Experiment Result

C.3.1 Document Type Definition

```
<!ELEMENT experimentResult (temp?, datavector+)>
<!ELEMENT temp (#PCDATA)>
<!ELEMENT datavector (#PCDATA)>
<!ATTLIST experimentResult
    lab          CDATA  #FIXED   "MIT Microelectronics Weblab"
    specversion  CDATA  #REQUIRED>
```

```
<!ATTLIST temp
    units        CDATA  #REQUIRED>
<!ATTLIST datavector
    name         CDATA  #REQUIRED
    units        CDATA  #REQUIRED>
```

C.3.2 Sample Document

```
<?xml version="1.0" encoding="utf-8" standalone="no" ?>
<!DOCTYPE experimentResult
    SYSTEM "http://weblab2.mit.edu/xml/experimentResult.dtd">
<experimentResult lab="MIT Microelectronics Weblab" specversion="0.1">
    <temp units="K">300</temp>
    <datavector name="VS" units="V">1,2,3,4,5</datavector>
    <datavector name="IS" units="I">6,7,8,9,0</datavector>
</experimentResult>
```

Appendix D

WeblabSim Client Document Formats

The following sections define the documents used by the WeblabSim client. For each document type, we give its schema and an example of a document conforming to the schema. A schema may be expressed as a Document Type Definition (DTD) [60] or as an XML Schema definition [79, 80, 81].

D.1 Laboratory Configuration

D.1.1 Document Type Definition

```
<!ELEMENT labConfiguration (device*)>
<!ATTLIST labConfiguration
    lab      CDATA  #FIXED   "MIT Device Simulation Weblab"
    version  CDATA  #REQUIRED>
<!ELEMENT device (name, description, imageURL, terminal+,
                  parameter*, maxDataPoints)>
<!ATTLIST device
    id    CDATA  #REQUIRED
    type  CDATA  #REQUIRED>
<!ELEMENT name (#PCDATA)>
<!ELEMENT description (#PCDATA)>
<!ELEMENT imageURL (#PCDATA)>
<!ELEMENT terminal (label, pixelLocation, maxVoltage, maxCurrent)>
```

```
<!ATTLIST terminal
    portType    (SMU | VSU | VMU)  #REQUIRED
    portNumber  CDATA              #REQUIRED>
<!ELEMENT label (#PCDATA)>
<!ELEMENT pixelLocation (x, y)>
<!ELEMENT x (#PCDATA)>
<!ELEMENT y (#PCDATA)>
<!ELEMENT maxDataPoints (#PCDATA)>
<!ATTLIST parameter
    name        CDATA           #REQUIRED
    units       CDATA           #REQUIRED
    description CDATA           #REQUIRED
    required    (true | false)  "true"
    default     CDATA           #IMPLIED>
```

D.1.2 Sample Document

```
<?xml version="1.0" encoding="utf-8" ?>
<labConfiguration lab="MIT Device Simulation Weblab" version="1.0-b1">
    <device id="5" type="nMOSFET">
        <name>6.012 nMOSFET</name>
        <description>Aren't nMOSFETs just swell?</description>
        <imageURL>http://weblab.mit.edu/5.0/icons/NMOS.gif</imageURL>
        <terminal portType="SMU" portNumber="2">
            <label>Gate</label>
            <pixelLocation>
                <x>121</x>
                <y>94</y>
            </pixelLocation>
            <maxVoltage>2.0</maxVoltage>
            <maxCurrent>0.1</maxCurrent>
        </terminal>
        <terminal portType="SMU" portNumber="3">
            <label>Source</label>
            <pixelLocation>
                <x>195</x>
                <y>156</y>
            </pixelLocation>
            <maxVoltage>5.0</maxVoltage>
            <maxCurrent>0.1</maxCurrent>
        </terminal>
        <terminal portType="SMU" portNumber="4">
            <label>Bulk</label>
            <pixelLocation>
                <x>235</x>
                <y>94</y>
            </pixelLocation>
            <maxVoltage>5.0</maxVoltage>
            <maxCurrent>0.1</maxCurrent>
        </terminal>
        <terminal portType="SMU" portNumber="1">
```

```
                <label>Drain</label>
                <pixelLocation>
                    <x>195</x>
                    <y>23</y>
                </pixelLocation>
                <maxVoltage>5.0</maxVoltage>
                <maxCurrent>0.1</maxCurrent>
            </terminal>
            <parameter name="L" units="m"
                description="channel length"
                default="1e-6" />
            <parameter name="W" units="m"
                description="channel width"
                default="10e-6" />
            <parameter name="KP" units="A/V^2"
                description="transconductance"
                default="2e-5" />
            <parameter name="VTO" units="V"
                description="zero-bias threshold voltage"
                default="0" />
            <parameter name="TOX" units="m"
                description="gate oxide thickness"
                default="10e-9" />
            <maxDataPoints>500</maxDataPoints>
        </device>
    </labConfiguration>
```

D.2 Simulation Specification

D.2.1 XML Schema

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema elementFormDefault="qualified" jaxb:version="1.0"
    targetNamespace="http://weblabsim.mit.edu"
    xmlns="http://weblabsim.mit.edu"
    xmlns:jaxb="http://java.sun.com/xml/ns/jaxb"
    xmlns:xs="http://www.w3.org/2001/XMLSchema">

    <xs:annotation>
        <xs:appinfo>
            <jaxb:schemaBindings>
                <jaxb:package name="edu.mit.weblabsim.domain.spec" />
            </jaxb:schemaBindings>
        </xs:appinfo>
    </xs:annotation>


    <!-- GLOBAL ELEMENT DECLARATIONS -->
```

```
<xs:element name="simulationSpecification">
    <xs:annotation>
        <xs:documentation>
            Describes a simulation in the MIT Device Simulation WebLab.
        </xs:documentation>
    </xs:annotation>
    <xs:complexType>
        <xs:sequence>
            <xs:element name="device" type="DeviceType" />
            <xs:element name="terminal" type="TerminalType"
                minOccurs="0" maxOccurs="unbounded" />
            <xs:element name="userDefinedFunction" type="UDFType"
                minOccurs="0" maxOccurs="unbounded" />
        </xs:sequence>
        <xs:attribute name="lab" type="xs:string" use="required"
            fixed="MIT Device Simulation WebLab" />
        <xs:attribute name="version" type="xs:string" use="required" />
    </xs:complexType>
</xs:element>


<!-- COMPLEX TYPE DEFINITIONS -->

<xs:complexType name="DeviceType">
    <xs:annotation>
        <xs:documentation>
            A device under test.
        </xs:documentation>
    </xs:annotation>
    <xs:sequence>
        <xs:element name="parameter" minOccurs="0" maxOccurs="unbounded">
            <xs:complexType>
                <xs:sequence>
                    <xs:element name="function" type="FunctionType" />
                </xs:sequence>
               <xs:attribute name="name" type="xs:string" use="required" />
           </xs:complexType>
        </xs:element>
    </xs:sequence>
    <xs:attribute name="id" type="xs:unsignedShort" use="required" />
</xs:complexType>

<xs:complexType name="TerminalType">
    <xs:annotation>
        <xs:documentation>
            A terminal connected to the device under test.
        </xs:documentation>
    </xs:annotation>
    <xs:sequence>
        <xs:element name="vname" type="VarNameType" />
        <xs:element name="iname" type="VarNameType" />
        <xs:element name="mode">
            <xs:simpleType>
                <xs:restriction base="xs:string">
```

```
                    <xs:enumeration value="V" />
                    <xs:enumeration value="I" />
                    <xs:enumeration value="COMM" />
                </xs:restriction>
            </xs:simpleType>
        </xs:element>
        <xs:element name="function" type="FunctionType" minOccurs="0" />
        <xs:element name="compliance" type="xs:double" minOccurs="0" />
    </xs:sequence>
    <xs:attribute name="portType" use="required" >
        <xs:simpleType>
            <xs:restriction base="xs:string">
                <xs:enumeration value="SMU" />
                <xs:enumeration value="VMU" />
                <xs:enumeration value="VSU" />
            </xs:restriction>
        </xs:simpleType>
    </xs:attribute>
    <xs:attribute name="portNumber" type="xs:unsignedShort" use="required" />
</xs:complexType>

<xs:complexType name="UDFType">
    <xs:annotation>
        <xs:documentation>
            A user defined function.
        </xs:documentation>
    </xs:annotation>
    <xs:sequence>
        <xs:element name="name" type="VarNameType" />
        <xs:element name="units" type="xs:string" />
        <xs:element name="body" type="xs:string" />
    </xs:sequence>
</xs:complexType>

<xs:complexType name="VarNameType">
    <xs:annotation>
        <xs:documentation>
            A variable name. This is a string, with an attribute that
            specifies whether or not the variable should be downloaded.
        </xs:documentation>
    </xs:annotation>
    <xs:simpleContent>
        <xs:extension base="xs:string">
            <xs:attribute name="download" type="xs:boolean" use="required" />
        </xs:extension>
    </xs:simpleContent>
</xs:complexType>

<xs:complexType name="FunctionType">
    <xs:annotation>
        <xs:documentation>
            A source function. This element specifies how the source
            (terminal or parameter) is to be programmed during the course
            of the simulation.
```

```
            </xs:documentation>
        </xs:annotation>
        <xs:choice>
            <xs:sequence>
                <xs:element name="scale">
                    <xs:simpleType>
                        <xs:restriction base="xs:string">
                            <xs:enumeration value="LIN" />
                            <xs:enumeration value="LOG10" />
                            <xs:enumeration value="LOG25" />
                            <xs:enumeration value="LOG50" />
                        </xs:restriction>
                    </xs:simpleType>
                </xs:element>
                <xs:element name="start" type="xs:double" />
                <xs:element name="stop" type="xs:double" />
                <xs:element name="step" type="xs:double" />
            </xs:sequence>
            <xs:sequence>
                <xs:element name="ratio" type="xs:double" />
                <xs:element name="offset" type="xs:double" />
            </xs:sequence>
            <xs:sequence>
                <xs:element name="value" type="xs:double" />
            </xs:sequence>
        </xs:choice>
        <xs:attribute name="type" use="required">
            <xs:simpleType>
                <xs:restriction base="xs:string">
                    <xs:annotation>
                        <xs:documentation>
                            The possible types are "CONS", "VAR{number}"
                            and "VAR{number}P".
                        </xs:documentation>
                    </xs:annotation>
                    <xs:pattern value="(CONS)|(VAR\d)|(VAR\dP)" />
                </xs:restriction>
            </xs:simpleType>
        </xs:attribute>
    </xs:complexType>
</xs:schema>
```

D.2.2 Sample Document

```
<?xml version="1.0" encoding="utf-8" ?>
<simulationSpecification xmlns="http://weblabsim.mit.edu"
    lab="MIT Device Simulation Weblab" version="1.0-b1">
    <device id="1">
        <parameter name="VTO">
            <function type="CONS">
            <value>0.5</value>
```

```
            </function>
        </parameter>
    </device>
    <terminal portType="SMU" portNumber="1">
        <vname download="true">VDS</vname>
        <iname download="true">IDS</iname>
        <mode>V</mode>
        <function type="VAR2">
            <scale>LIN</scale>
            <start>0.05</start>
            <stop>0.5</stop>
            <step>0.1</step>
        </function>
        <compliance>0.1</compliance>
    </terminal>
    <terminal portType="SMU" portNumber="2">
        <vname download="true">VG</vname>
        <iname download="true">IG</iname>
        <mode>V</mode>
        <function type="VAR1">
            <scale>LIN</scale>
            <start>0.0</start>
            <stop>4.5</stop>
            <step>0.05</step>
        </function>
        <compliance>0.1</compliance>
    </terminal>
    <terminal portType="SMU" portNumber="3">
        <vname download="true">VGRD</vname>
        <iname download="true">IGRD</iname>
        <mode>COMM</mode>
        <compliance>0.1</compliance>
    </terminal>
    <terminal portType="SMU" portNumber="4">
        <vname download="true">VB</vname>
        <iname download="true">IB</iname>
        <mode>V</mode>
        <function type="VAR1P">
            <offset>4.5</offset>
            <ratio>0.05</ratio>
        </function>
        <compliance>0.1</compliance>
    </terminal>
    <userDefinedFunction>
        <name download="true">SQRTID</name>
        <units>A</units>
        <body>SQRT(IDS)</body>
    </userDefinedFunction>
</simulationSpecification>
```

D.3 Simulation Results

D.3.1 XML Schema

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema elementFormDefault="qualified"
    targetNamespace="http://weblabsim.mit.edu/schemas/iof/1.0/"
    xmlns="http://weblabsim.mit.edu/schemas/iof/1.0/"
    xmlns:xs="http://www.w3.org/2001/XMLSchema">

    <!-- GLOBAL ELEMENTS -->

    <xs:element name="simulationResult">
        <xs:annotation>
            <xs:documentation>
                Describes the result of a simulation in the MIT Device
                Simulation WebLab.
            </xs:documentation>
        </xs:annotation>
        <xs:complexType>
            <xs:sequence>
                <xs:element name="temp" minOccurs="0">
                    <complexType>
                        <xs:simpleContent>
                            <xs:extension base="xs:double">
                                <xs:attribute name="units" type="xs:string"/>
                            </xs:extension>
                        </xs:simpleContent>
                    </complexType>
                </xs:element>
                <xs:element name="datavector" type="VariableType"
                    minOccurs="0" maxOccurs="unbounded" />
            </xs:sequence>
            <xs:attribute name="lab" type="xs:string" use="required"
                fixed="MIT Device Simulation WebLab" />
            <xs:attribute name="version" type="xs:string" use="required" />
        </xs:complexType>
    </xs:element>

    <!-- TYPE DEFINITIONS -->

    <xs:complexType name="VariableType">
        <xs:simpleContent>
            <xs:extension base="ValuesType">
                <xs:attribute name="name" type="NameType" use="required"/>
                <xs:attribute name="units" type="NameType" use="required"/>
        </xs:simpleContent>
    </xs:complexType>

    <xs:simpleType name="NameType">
        <xs:annotation>
```

```
            <xs:documentation>
                A name is a sequence of numbers, letters, or underscores.
            </xs:documentation>
        </xs:annotation>
        <xs:restriction base="xs:string">
            <xs:pattern value="[0-9A-Za-z_]+"/>
        </xs:restriction>
    </xs:simpleType>

    <xs:simpleType name="ValuesType">
        <xs:annotation>
            <xs:documentation>
                A sequence of floating-point values (double precision).
            </xs:documentation>
        </xs:annotation>
        <xs:list itemType="xs:double"/>
    </xs:simpleType>
</xs:schema>
```

D.3.2 Sample Document

```
<?xml version="1.0" encoding="utf-8" ?>
<simulationResult lab="MIT Device Simulation WebLab" version="1.0-b1">
    <temp units="K">300</temp>
    <datavector name="VS" units="V">1 2 3 4 5</datavector>
    <datavector name="IS" units="I">6 7 8 9 0</datavector>
</simulationResult>
```

Appendix E

Worker Server Document Formats

E.1 Web Service Interface

The following document is a description of the worker server web service interface, written in the Web Service Definition Language (WSDL) [62]. The definition assumes a worker server located at http://weblab2.mit.edu:8080/worker/services/WorkerService. If the service is deployed at another URL, then the location attribute of the wsdl-soap:address element must be modified accordingly.

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions targetNamespace="http://weblabsim.mit.edu/labserver/worker"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:apachesoap="http://xml.apache.org/xml-soap"
    xmlns:impl="http://weblabsim.mit.edu/labserver/worker"
    xmlns:intf="http://weblabsim.mit.edu/labserver/worker"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">
 <wsdl:types>
  <schema targetNamespace="http://weblabsim.mit.edu/labserver/
      xmlns="http://www.w3.org/2001/XMLSchema"
      xmlns:wls="http://weblabsim.mit.edu">
   <import namespace="http://weblabsim.mit.edu"/>
   <element name="submit">
    <complexType>
     <sequence>
      <element ref="wls:simulationSpecification" />
     </sequence>
    </complexType>
```

```
  <element>
  <element name="submitReturn">
   <complexType>
    <sequence>
     <element ref="wls:simulationResult" />
    </sequence>
   </complexType>
  </element>
  <element name="ping">
   <complexType />
  </element>
  <element name="pingReturn">
   <complexType />
  </element>
 </schema>
</wsdl:types>
  <wsdl:message name="submitResponse">
     <wsdl:part element="impl:submitReturn" name="submitReturn"/>
  </wsdl:message>
  <wsdl:message name="pingResponse">
     <wsdl:part element="impl:pingReturn" name="pingReturn"/>
  </wsdl:message>
  <wsdl:message name="submitRequest">
     <wsdl:part element="impl:submit" name="part"/>
  </wsdl:message>
  <wsdl:message name="pingRequest">
     <wsdl:part element="impl:ping" name="part"/>
  </wsdl:message>
  <wsdl:portType name="Worker">
     <wsdl:operation name="submit">
        <wsdl:input message="impl:submitRequest" name="submitRequest"/>
        <wsdl:output message="impl:submitResponse" name="submitResponse"/>
     </wsdl:operation>
     <wsdl:operation name="ping">
        <wsdl:input message="impl:pingRequest" name="pingRequest"/>
        <wsdl:output message="impl:pingResponse" name="pingResponse"/>
     </wsdl:operation>
  </wsdl:portType>
  <wsdl:binding name="WorkerSoapBinding" type="impl:Worker">
     <wsdlsoap:binding style="document"
         transport="http://schemas.xmlsoap.org/soap/http"/>
     <wsdl:operation name="submit">
        <wsdlsoap:operation soapAction=""/>
        <wsdl:input name="submitRequest">
           <wsdlsoap:body use="literal"/>
        </wsdl:input>
        <wsdl:output name="submitResponse">
           <wsdlsoap:body use="literal"/>
        </wsdl:output>
     </wsdl:operation>
    <wsdl:operation name="ping">
        <wsdlsoap:operation soapAction=""/>
        <wsdl:input name="pingRequest">
           <wsdlsoap:body use="literal"/>
```

```
            </wsdl:input>
            <wsdl:output name="pingResponse">
               <wsdlsoap:body use="literal"/>
            </wsdl:output>
         </wsdl:operation>
      </wsdl:binding>
      <wsdl:service name="WorkerService">
         <wsdl:port binding="impl:WorkerSoapBinding" name="Worker">
            <wsdlsoap:address
   location="http://weblab2.mit.edu:8080/worker/services/WorkerService"/>
         </wsdl:port>
      </wsdl:service>
   </wsdl:definitions>
```

E.2 Intermediate Input Format

E.2.1 XML Schema

```
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<xs:schema elementFormDefault="qualified" jaxb:version="1.0"
    targetNamespace="http://weblabsim.mit.edu/schemas/iif/1.0/"
    xmlns="http://weblabsim.mit.edu/schemas/iif/1.0/"
    xmlns:jaxb="http://java.sun.com/xml/ns/jaxb"
    xmlns:xs="http://www.w3.org/2001/XMLSchema">

    <xs:annotation>
        <xs:appinfo>
            <jaxb:schemaBindings>
                <jaxb:package name="edu.mit.weblabsim.input" />
            </jaxb:schemaBindings>
        </xs:appinfo>
    </xs:annotation>


    <!-- GLOBAL ELEMENT DECLARATIONS -->

    <xs:element name="experiment">
        <xs:annotation>
            <xs:documentation>
                The root element. An experiment in the Device Simulation WebLab
                defines a device simulation and the quantities to gather/compute
                based on the results of the simulation.
            </xs:documentation>
        </xs:annotation>
        <xs:complexType>
            <xs:sequence>
                <xs:element name="simulation" type="SimulationType"/>
                <xs:element name="variables">
                    <xs:complexType>
```

```
                    <xs:sequence>
                        <xs:element name="variable" type="VariableType"
                            maxOccurs="unbounded" minOccurs="0" />
                    </xs:sequence>
                </xs:complexType>
            </xs:element>
        </xs:sequence>
    </xs:complexType>
</xs:element>

<xs:element name="element" type="ElementType" />

<!-- SIMPLE TYPE DECLARATIONS -->

<xs:simpleType name="NodeType">
    <xs:annotation>
        <xs:documentation>
            Represents a node in a circuit. As in SPICE, nodes are specified
            by non-negative integers (0, 1, 2,...), where node 0 is the
            circuit ground node.
        </xs:documentation>
    </xs:annotation>
    <xs:restriction base="xs:int">
        <xs:minInclusive value="0" />
    </xs:restriction>
</xs:simpleType>

<xs:simpleType name="NodeListType">
    <xs:list itemType="NodeType"/>
</xs:simpleType>

<xs:simpleType name="DeclaredNodeListType">
    <xs:annotation>
        <xs:documentation>
            A declared node list type is a list of nodes, none of which can
            be the ground node.
        </xs:documentation>
    </xs:annotation>
    <xs:list>
        <xs:simpleType>
            <xs:restriction base="NodeType">
                <xs:minInclusive value="1"/>
            </xs:restriction>
        </xs:simpleType>
    </xs:list>
</xs:simpleType>

<xs:simpleType name="NameType">
    <xs:annotation>
        <xs:documentation>
            A name is a sequence of numbers, letters, or underscores.
        </xs:documentation>
    </xs:annotation>
    <xs:restriction base="xs:string">
```

```
        <xs:pattern value="[0-9A-Za-z_]+"/>
    </xs:restriction>
</xs:simpleType>

<xs:simpleType name="MeterType">
    <xs:annotation>
        <xs:appinfo>
            <jaxb:typesafeEnumClass>
                <jaxb:typesafeEnumMember name="VOLTAGE" value="voltage" />
                <jaxb:typesafeEnumMember name="CURRENT" value="current" />
                <jaxb:typesafeEnumMember name="PORT" value="port" />
            </jaxb:typesafeEnumClass>
        </xs:appinfo>
    </xs:annotation>
    <xs:restriction base="xs:string">
        <xs:enumeration value="voltage"/>
        <xs:enumeration value="current"/>
        <xs:enumeration value="port"/>
    </xs:restriction>
</xs:simpleType>

<!-- COMPLEX TYPE DEFINITIONS -->

<xs:complexType name="KeyValueType">
    <xs:annotation>
        <xs:documentation>
            A key-value pair associated a name (see NameType) with a string
            value.
        </xs:documentation>
    </xs:annotation>
    <xs:attribute name="name" type="NameType" use="required"/>
    <xs:attribute name="value" type="xs:string" use="required"/>
    <xs:attribute name="unit" type="xs:string" use="optional" default="" />
</xs:complexType>

<xs:complexType name="ElementType">
    <xs:sequence>
        <xs:element name="param" type="KeyValueType"
            minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
    <xs:attribute name="id" type="NameType" use="required"/>
    <xs:attribute name="type" type="xs:QName" use="required">
        <xs:annotation>
            <xs:documentation>
                The type of a circuit element is a qualified name, consisting
                of a namespace and a local part. The namespace identifies the
                general family of devices/definitions, and the local part
                names the particular device within that family. For example,
                {urn:devices:weblabsim:1.0}IndependentVoltageSource
                refers to the independent voltage source in the WeblabSim
                standard device family.
            </xs:documentation>
        </xs:annotation>
    </xs:attribute>
```

```
    <xs:attribute name="nodes" type="NodeListType" use="required"/>
</xs:complexType>

<xs:complexType name="CircuitType">
    <xs:sequence>
        <xs:element ref="element" maxOccurs="unbounded"/>
    </xs:sequence>
    <xs:attribute name="nodes" type="DeclaredNodeListType" use="required"/>
</xs:complexType>

<xs:complexType name="VoltageMeterType">
    <xs:attribute name="name" type="NameType" use="required"/>
    <xs:attribute name="node" type="NodeType" use="required"/>
    <xs:attribute name="type" type="MeterType" fixed="voltage"/>
</xs:complexType>
<xs:complexType name="CurrentMeterType">
    <xs:attribute name="name" type="NameType" use="required"/>
    <xs:attribute name="element" type="NameType" use="required"/>
    <xs:attribute name="node" type="NodeType" use="required"/>
    <xs:attribute name="type" type="MeterType" fixed="current"/>
</xs:complexType>
<xs:complexType name="PortMeterType">
    <xs:attribute name="name" type="NameType" use="required"/>
    <xs:attribute name="element" type="NameType" use="required"/>
    <xs:attribute name="port" type="NameType" use="required"/>
    <xs:attribute name="type" type="MeterType" fixed="port"/>
</xs:complexType>

<xs:complexType name="SweepType">
    <xs:attribute name="element" type="NameType" use="required"/>
    <xs:attribute name="param" type="NameType" use="required"/>
    <xs:attribute name="type" use="required">
        <xs:simpleType>
            <xs:restriction base="xs:string">
                <xs:enumeration value="linear"/>
                <xs:enumeration value="logarithmic"/>
            </xs:restriction>
        </xs:simpleType>
    </xs:attribute>
    <xs:attribute name="start" type="xs:decimal" use="required"/>
    <xs:attribute name="stop" type="xs:decimal" use="required"/>
    <xs:attribute name="step" type="xs:decimal" use="optional">
        <xs:annotation>
            <xs:documentation>
                If "type" equals "linear", the "step" attribute is required,
                and is the size of the linear step to take while sweeping
                from "start" to "stop". If type equals "logarithmic", the
                "step" attribute is ignored.
            </xs:documentation>
        </xs:annotation>
    </xs:attribute>
    <xs:attribute name="pointsPerDecade" use="optional">
        <xs:annotation>
            <xs:documentation>
```

```
                    If "type" equals "logarithmic", the "pointsPerDecade"
                    attribute is required, and is the number of points per
                    decade to insert between "start" and "stop". If "type"
                    equals "linear", the "pointsPerDecade" attribute is
                    ignored.
                </xs:documentation>
            </xs:annotation>
            <xs:simpleType>
                <xs:restriction base="xs:decimal">
                    <xs:minInclusive value="1"/>
                </xs:restriction>
            </xs:simpleType>
        </xs:attribute>
    </xs:complexType>

    <xs:complexType name="VariableType">
        <xs:attribute name="name" type="NameType" use="required"/>
        <xs:attribute name="unit" type="xs:string" use="required"/>
        <xs:attribute name="expr" type="xs:string" use="optional"/>
        <xs:attribute name="download" type="xs:boolean" use="required"/>
    </xs:complexType>

    <xs:complexType name="SimulationType">
        <xs:sequence>
            <xs:element name="circuit" type="CircuitType"/>
            <xs:element name="measures">
                <xs:complexType>
                    <xs:choice minOccurs="0" maxOccurs="unbounded">
                        <xs:element name="voltageMeter" type="VoltageMeterType"/>
                        <xs:element name="currentMeter" type="CurrentMeterType"/>
                        <xs:element name="portMeter" type="PortMeterType"/>
                    </xs:choice>
                </xs:complexType>
            </xs:element>
            <xs:element name="actions">
                <xs:complexType>
                    <xs:sequence>
                        <xs:element name="sweep" type="SweepType"
                            minOccurs="0" maxOccurs="unbounded"/>
                    </xs:sequence>
                </xs:complexType>
            </xs:element>
            <xs:element name="options">
                <xs:complexType>
                    <xs:sequence>
                        <xs:element name="option" type="KeyValueType"
                            minOccurs="0" maxOccurs="unbounded"/>
                    </xs:sequence>
                </xs:complexType>
            </xs:element>
        </xs:sequence>
        <xs:attribute name="simulator" type="xs:anyURI" use="required" />
    </xs:complexType>
</xs:schema>
```

E.2.2 Sample Document

```
<?xml version="1.0" encoding="utf-8"?>
<experiment xmlns="http://weblabsim.mit.edu/schemas/iif/1.0/">
    <simulation simulator="urn:simulator:WinSpice/1.05.04"
        xmlns:ws="urn:simulator:WinSpice/1.05.04">
        <circuit nodes="1 2">
            <element id="device" type="ws:Resistor" nodes="1 2">
                <param name="R" value="5.0" unit="OHM" />
            </element>
            <element id="unit1" nodes="1 0" type="ws:IndependentVoltageSource">
                <param name="VALUE" value="0" unit="V" />
            </element>
            <element id="unit2" nodes="2 0" type="ws:IndependentVoltageSource">
                <param name="VALUE" value="0" unit="V" />
            </element>
        </circuit>
        <measures>
            <voltageMeter name="V1" node="1" />
            <currentMeter name="I1" element="V1" node="1" />
            <voltageMeter name="V2" node="2" />
            <currentMeter name="I2" element="V2" node="2" />
            <portMeter name="R" element="R1" port="value" />
        </measures>
        <actions>
            <sweep element="V1" param="VALUE" type="linear"
                start="0" stop="1" step="0.1" />
            <sweep element="V2" param="VALUE" type="linear"
                start="0" stop="1" step="0.1" />
        </actions>
        <options>
            <option name="TEMP" value="25" />
        </options>
    </simulation>
    <variables>
        <variable name="V1" unit="V" expr="V1" download="true" />
        <variable name="I1" unit="I" expr="I1" download="false" />
        <variable name="V2" unit="V" expr="V2" download="true" />
        <variable name="I2" unit="I" expr="I2" download="false" />
        <variable name="R" unit="OHM" expr="R" download="true" />
        <variable name="VR" unit="V" expr="V2 - V1" download="true"/>
    </variables>
</experiment>
```

E.3 Intermediate Output Format

E.3.1 XML Schema

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema elementFormDefault="qualified" jaxb:version="1.0"
    targetNamespace="http://weblabsim.mit.edu/schemas/iof/1.0/"
    xmlns="http://weblabsim.mit.edu/schemas/iof/1.0/"
    xmlns:jaxb="http://java.sun.com/xml/ns/jaxb"
    xmlns:xs="http://www.w3.org/2001/XMLSchema">

    <xs:annotation>
        <xs:appinfo>
            <jaxb:globalBindings>
                <jaxb:javaType name="java.lang.Double"  xmlType="xs:double"
            printMethod="edu.mit.weblabsim.output.DecimalPrinter.printDouble" />
            </jaxb:globalBindings>
            <jaxb:schemaBindings>
                <jaxb:package name="edu.mit.weblabsim.output" />
            </jaxb:schemaBindings>
        </xs:appinfo>
    </xs:annotation>

    <!-- GLOBAL ELEMENTS -->

    <xs:element name="result">
        <xs:complexType>
            <xs:sequence>
                <xs:element name="variable" type="VariableType"
                    minOccurs="0" maxOccurs="unbounded" />
            </xs:sequence>
        </xs:complexType>
    </xs:element>

    <!-- TYPE DEFINITIONS -->

    <xs:complexType name="VariableType">
        <xs:annotation>
            <xs:appinfo>
                <jaxb:property>
                    <jaxb:javadoc><![CDATA[
    Gets the value of this variable, which is a list of
    floating-point numbers. The list returned may be cast to
    an instance of <code>it.unimi.dsi.fastutil.doubles.DoubleList</code>.
    This allows direct manipulation of <code>double</code>s
    for better speed and memory performance.

                    ]]></jaxb:javadoc>
                </jaxb:property>
            </xs:appinfo>
        </xs:annotation>
```

```
        <xs:simpleContent>
            <xs:extension base="ValuesType">
                <xs:attribute name="name" type="NameType" use="required"/>
                <xs:attribute name="unit" type="NameType" use="required"/>
                <xs:attribute name="dim" use="required">
                    <xs:annotation>
                        <xs:appinfo>
                            <jaxb:property>
                                <jaxb:javadoc><![CDATA[
    Gets the dimensions of this variable, which is a list of
    integers corresponding to the number of elements in each
    dimension. The list returned may be cast to
    an instance of <code>it.unimi.dsi.fastutil.ints.IntList</code>.
    This allows direct manipulation of <code>int</code>s
    for better speed and memory performance.

                                ]]></jaxb:javadoc>
                            </jaxb:property>
                        </xs:appinfo>
                    </xs:annotation>
                    <xs:simpleType>
                        <xs:list itemType="xs:int"/>
                    </xs:simpleType>
                </xs:attribute>
            </xs:extension>
        </xs:simpleContent>
    </xs:complexType>

    <xs:simpleType name="NameType">
        <xs:annotation>
            <xs:documentation>
                A name is a sequence of numbers, letters, or underscores.
            </xs:documentation>
        </xs:annotation>
        <xs:restriction base="xs:string">
            <xs:pattern value="[0-9A-Za-z_]+"/>
        </xs:restriction>
    </xs:simpleType>

    <xs:simpleType name="ValuesType">
        <xs:annotation>
            <xs:documentation>
                A sequence of floating-point values (double precision).
            </xs:documentation>
        </xs:annotation>
        <xs:list itemType="xs:double"/>
    </xs:simpleType>
</xs:schema>
```

E.3.2 Sample Document

```
<?xml version="1.0" encoding="utf-8"?>
<result xmlns="http://weblabsim.mit.edu/schemas/iof/1.0/">
    <variable name="V1" dim="2 2" unit="V">+1.0000000E+000 +1.2000000E+000
        +1.3000000E+000 +1.4000000E+000</variable>
    <variable name="RD" dim="2 2" unit="OHM">+5.0000000E+000 +5.0000000E+000
        +5.0000000E+000 +5.0000000E+000</variable>
</result>
```

Bibliography

[1] M. Smith, "Winspice," Jan. 2004. [Online]. Available: http://www.winspice.com

[2] L. W. Nagel, "SPICE2: A computer program to simulate semiconductor circuits," Univ. of California, Electronic Research Laboratory," ERL-M520, 1975.

[3] Synopsys Inc., "HSPICE™," 700 East Middlefield Road, Mountain View, CA 94043, USA.

[4] Cadence Design Systems Inc., "PSpice®A/D," 2655 Seely Avenue, San Jose, CA 95134, USA.

[5] Altium Limited, "Circuitmaker™ 2000," 12A Rodborough Road, French Forest NSW 2086, Australia.

[6] Electronics Workbench Inc., "MultiSim™," 908 Niagara Falls Boulevard, Suite #068 North Tonawanda, NY 14120-2060, USA.

[7] MSC Software Inc., "Working Model™," 66 Bovet Road, Suite 200, San Mateo, CA 94402, USA.

[8] K. D. Forbus, P. B. Whalley, J. O. Everett, L. Ureel, M. Brokowski, J. Baher, and S. E. Kuehne, "CyclePad: An articulate virtual laboratory for engineering thermodynamics," *Artificial Intelligence*, vol. 114, pp. 297–347, 1999. [Online]. Available: citeseer.ist.psu.edu/white90causal.html

[9] National Semiconductor Corp., "WEBBENCH Electrical Simulator," 2900 Semiconductor Drive, Santa Clara, CA 95052, USA. [Online]. Available: http://www.national.com/appinfo/power/webench/elec_sim.html

[10] Intersil Corp., "iSim Design Simulation Tool," 675 Trade Zone Blvd., Milpitas, CA 95035, USA. [Online]. Available: http://www.intersil.com/isim/

[11] I. van Rienen, "DigSim: A digital schematic editor and simulator," 1995. [Online]. Available: http://www.iwans.net

[12] N. S. Ghazal, "DigSim, WELD project version," 1996. [Online]. Available: http://www-cad.eecs.berkeley.edu/weld/yoyodyne_demo_docs/DigSim/DigSim.html

[13] M. Karweit, "A virtual engineering/science laboratory course," Feb. 2000. [Online]. Available: http://www.jhu.edu/ virtlab/virtlab.html

[14] University of Oxford Virtual Reality Group, "Virtual Chemistry," Aug. 2003. [Online]. Available: http://www.chem.ox.ac.uk/vrchemistry/

[15] J. A. del Alamo, L. Brooks, C. McLean, J. Hardison, G. Mishuris, V. Chang, and L. Hui, "The MIT Microelectronics WebLab: a web-enabled remote laboratory for microelectronic device characterization," in *World Congress on Networked Learning in a Global Environment*, Berlin, Germany, 2002.

[16] ——, "Educational experiments with an online microelectronics laboratory," in *International Conference on Engineering Education 2002*, Manchester, UK, 2002.

[17] ——, "MIT Microelectronics WebLab," in *Lab on the Web: Running Real Electronics Experiments via the Internet*, T. A. Fjeldly and M. S. Shur, Eds. New Jersey: Wiley-IEEE, 2003, pp. 49–87.

[18] J. A. del Alamo, V. Chang, J. Hardison, D. Zych, and L. Hui, "An online microelectronics device characterization laboratory with a circuit-like user interface," in *International Conference on Engineering Education 2003*, Valencia, Spain, 2003.

[19] K. O. Jeppson, P. Lundgren, J. A. del Alamo, J. L. Hardison, and D. Zych, "Sharing online laboratories and their components: A new learning experience," in *5th European Workshop on Microelectronics Education*, Lausanne, Switzerland, Apr. 2004.

[20] V. Chang, "A new user interface for the MIT Microelectronics WebLab," MIT Microsystems Technology Laboratories," Annual Report, 2002.

[21] Y. Lin, "A collaboration system and a graphical interface for the MIT Microelectronics WebLab," M.Eng. Thesis, Massachusetts Institute of Technology, May 2002.

[22] V. J. Harward, J. A. del Alamo, V. S. Choudhary, K. deLong, J. L. Hardison, S. R. Lerman, J. Northridge, D. Talavera, C. Varadharajan, S. Wang, K. Yehia, and D. Zych, "iLab: A scalable architecture for sharing online experiments," in *International Conference on Engineering Education 2004*, Gainesville, Florida, 2004.

[23] D. N. Jackson, "Alloy: A lightweight object modelling notation," *ACM Transactions on Software Engineering and Methodology*, vol. 11, no. 2, pp. 256–290, Apr. 2002.

[24] A. S. Sedra and K. C. Smith, *Microelectronic Circuits*, 4th ed. New York, NY: Oxford University Press, 1998.

[25] *Star-Hspice Manual 2001.2*, Avant! Corporation, Fremont, CA, June 2001.

[26] M. Smith, *WinSpice3 User's Manual*, Feb. 2004.

[27] D. N. Jackson, "Lecture notes on software design," Fall 2000, lecture notes for 6.170: Software Engineering Laboratory class at MIT. [Online]. Available: http://sdg.lcs.mit.edu/ dnj/publications/fall00-lectures.pdf

[28] Maplesoft, "Maple™," 615 Kumpf Drive, Waterloo, Ontario, Canada N2V 1K8.

[29] "ilab: Remote online laboratories," Dec. 2003. [Online]. Available: http://icampus.mit.edu/projects/iLab.shtml

[30] M. Gudgin, M. Hadley, N. Mendelsohn, J.-J. Moreau, and H. F. Nielsen, eds., *SOAP Version 1.2 Part 1: Messaging Framework*, World Wide Web Consortium, 24 June 2003. [Online]. Available: http://www.w3.org/TR/soap12-part1/

[31] ——, *SOAP Version 1.2 Part 2: Adjuncts*, World Wide Web Consortium, 24 June 2003. [Online]. Available: http://www.w3.org/TR/soap12-part2/

[32] A. S. Tanenbaum, *Modern Operating Systems.* Upper Saddle River, NJ: Prentice Hall, 2001.

[33] B. Kemme, "Database replication for clusters of workstations," PhD Dissertation, Swiss Federal Institute of Technology Zurich, 2000.

[34] J. Postel, ed., *RFC 791: Internet Protocol (DARPA Internet Program Protocol Specification)*, Internet Engineering Task Force, Sept. 1981. [Online]. Available: http://ietf.org/rfc/rfc791.txt

[35] S. Deering and R. Hinden, *RFC 1883: Internet Protocol, Version 6 (IPv6) Specification*, Internet Engineering Task Force, Dec. 1995. [Online]. Available: http://ietf.org/rfc/rfc1883.txt

[36] P. Mockapetris, *RFC 1034: Domain Names — concepts and facilities*, Internet Engineering Task Force, Nov. 1987. [Online]. Available: http://ietf.org/rfc/rfc1034.txt

[37] V. Chang, "Remote collaboration in WebLab – an online laboratory," M.Eng. Thesis, Massachusetts Institute of Technology, May 2001.

[38] Consortium ObjectWeb, "kSOAP: Open source SOAP for the kVM," INRIA-ZIRST, 655 avenue de l'Europe, Montbonnot, 38334 SAINT-ISMIER Cedex, France. [Online]. Available: http://ksoap.objectweb.org/

[39] The Unicode Consortium, *The Unicode Standard, Version 4.0.* Reading, MA, USA: Addison-Wesley, 2003, includes CD-ROM. [Online]. Available: http://www.unicode.org/versions/Unicode4.0.0/

[40] A. N. S. Institute, *Information Systems - Coded Character Sets - 7-Bit American National Standard Code for Information Interchange (7-Bit ASCII)*, 2002.

[41] A. Deitsch and D. Czarnecki, *Java Internationalization.* Cambridge, MA: O'Reilly, Mar. 2001.

[42] Massachusetts Institute of Technology, "MIT OpenCourseWare: OCW Home," http://ocw.mit.edu/index.html, 27 Aug. 2004.

[43] "OpenCourseWare spreading worldwide," *MIT Tech Talk*, vol. 48, no. 26, 5 May 2004. [Online]. Available: http://web.mit.edu/newsoffice/2004/ocw-0505.html

[44] Sun Microsystems Inc., "Java™ Plug-in Technology," 4150 Network Circle, Santa Clara, CA 95054, USA. [Online]. Available: http://java.sun.com/products/plugin/

[45] ——, *Supported System Configurations in Java™ 2 Platform v1.4.2*, July 2004. [Online]. Available: http://java.sun.com/j2se/1.4.2/system-configurations.html

[46] D. Alur, D. Malks, and J. Crupi, *Core J2EE Patterns: Best Practices and Design Strategies*, 2nd ed. Upper Saddle River, NJ: Prentice Hall, 2003.

[47] Sleepycat Software, "Berkeley DB Java Edition," 118 Tower Road, Lincoln, MA 01773, USA. [Online]. Available: http://www.sleepycat.com/products/je.shtml

[48] IBM Corporation, "IBM Cloudscape V10.0," 1133 Westchester Avenue, White Plains, NY 10604, USA. [Online]. Available: http://www-306.ibm.com/software/data/cloudscape/

[49] T. Berners-Lee, R. Fielding, and L. Masinter, *RFC 2396: Uniform Resource Identifiers (URI) Generic Syntax*, Internet Engineering Task Force, Aug. 1998. [Online]. Available: http://ietf.org/rfc/rfc2396.txt

[50] T. Berners-Lee, R. Fielding, and M. McCahill, *RFC 1783: Uniform Resource Locators (URL)*, Internet Engineering Task Force, Dec. 1994. [Online]. Available: http://ietf.org/rfc/rfc1738.txt

[51] Microsoft Corporation, "Microsoft SQL Server™ 2000," One Microsoft Way, Redmond, WA 98052, USA.

[52] MySQL AB, "MySQL® Database Server," Bangårdgatan 8, S-753 20 Uppsala, Sweden. [Online]. Available: http://www.mysql.com/products/mysql/

[53] K. Delaney, *Inside Microsoft SQL Server™ 2000.* Redmond, WA: Microsoft Press, 2001.

[54] S. White and M. Hapner, *JDBC™ 2.1 API*, Sun Microsystems Inc., 5 Oct. 1999. [Online]. Available: http://java.sun.com/products/jdbc/download.html#corespec21

[55] J. Ellis, L. Ho, and M. Fisher, *JDBC^{TM} 3.0 Specification*, Sun Microsystems Inc., 1 Dec. 2001. [Online]. Available: http://java.sun.com/products/jdbc/download.html#corespec30

[56] I. O. for Standardization, *ISO/IEC 9075:1999, Database Language SQL: Parts 1 (Framework), 2 (Foundation), and 5 (Bindings)*, 1999.

[57] Y.-T. Lau, *The Art of Objects: Object-Oriented Design and Architecture*, ser. The Addison-Wesley Object Technology Series. Boston, MA: Addison-Wesley Professional, 2001.

[58] "Hibernate." [Online]. Available: http://www.hibernate.org

[59] Apache Software Foundation, "WebServices – Axis," 19 Aug. 2004. [Online]. Available: http://ws.apache.org/axis/

[60] T. Bray, J. Paoli, C. M. Spergerg-McQueen, E. Maler, F. Yergeau, J. Cowan, and J. Cowan, *Extensible Markup Language (XML) 1.1*, World Wide Web Consortium, 4 Feb. 2004. [Online]. Available: http://www.w3.org/TR/xml11/

[61] Apache Software Foundation, *Axis Architecture Guide*, 13 July 2004. [Online]. Available: http://ws.apache.org/axis/java/architecture-guide.html

[62] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana, *Web Services Description Language (WSDL) 1.1*, World Wide Web Consortium, 15 Mar. 2001. [Online]. Available: http://www.w3.org/TR/2001/NOTE-wsdl-20010315

[63] J. Clark, ed., *XSL Transformations (XSLT)*, World Wide Web Consortium, 16 Nov. 1999. [Online]. Available: http://www.w3.org/TR/1999/REC-xslt-19991116

[64] J. Fialli and S. Vajjhala, eds., *The Java™ Architecture for XML Binding (JAXB)*, Sun Microsystems Inc., 8 Jan. 2003. [Online]. Available: http://java.sun.com/xml/downloads/jaxb.html

[65] Apache Software Foundation, *Axis Reference Guide*, 13 July 2004. [Online]. Available: http://ws.apache.org/axis/java/reference.html

[66] R. Irani and S. J. Basha, *AXIS: Next Generation Java SOAP.* Peer Information, May 2002.

[67] G. Viedma, I. Dancy, and K. Lundberg, "6.302 iLab homepage," http://web.mit.edu/6.302/www/weblab/, Aug. 2004.

[68] G. Viedma, "Design and implementation of a feedback systems web laboratory prototype," Advanced Undergraduate Project, Massachusetts Institute of Technology, May 2004. [Online]. Available: http://web.mit.edu/6.302/www/weblab/docs/viedma.pdf

[69] L. G. DeMichiel, *Enterprise JavaBeans™ Specification, Version 2.1*, Sun Microsystems Inc., 12 Nov. 2003. [Online]. Available: http://java.sun.com/products/ejb/docs.html

[70] T. H. Corment, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*, 1st ed. Cambridge, MA: The MIT Press, 1999.

[71] B. Shannon, ed., "Java™ 2 Platform, Enterprise Edition (J2EE™) specification," 24 Nov. 2003. [Online]. Available: http://java.sun.com/j2ee/j2ee-1_4-fr-spec.pdf

[72] Sun Microsystems Inc., "Sun Java™ System Application Server Platform Edition 8 Update 1," 4150 Network Circle, Santa

Clara, CA 95054, USA, June 2004. [Online]. Available: http://wwws.sun.com/software/products/appsrvr_pe/index.html

[73] ——, *Sun Java^{TM} System Application Server Platform Edition 8 Update 1 Release Notes*, June 2004.

[74] J. Hardison, "MIT Microelectronics WebLab v. 4.2: Addition of a thermometer plus a new management system," MIT Microsystems Technology Laboratories," Annual Report, 2002.

[75] B. Joy, G. Steele, J. Gosling, and G. Bracha, *Java^{TM}Language Specification*, 2nd ed. Boston, MA: Addison-Wesley Professional, 2000.

[76] IEEE Task P754, *IEEE Standard for Binary Floating-Point Arithmetic*, IEEE, New York, 12 Aug. 1985.

[77] J. Harward, "Service broker/lab server API," 29 Oct. 2003, internal specification document.

[78] D. Zych and J. Harward, "Client/service broker API," 16 Sept. 2003, internal specification document.

[79] D. C. Fallside, ed., *XML Schema Part 0: Primer*, World Wide Web Consortium, 2 May 2001. [Online]. Available: http://www.w3.org/TR/xmlschema-0/

[80] H. S. Thompson, D. Beech, M. Maloney, and N. Mendelsohn, eds., *XML Schema Part 1: Structures*, World Wide Web Consortium, 2 May 2001. [Online]. Available: http://www.w3.org/TR/xmlschema-1/

[81] P. V. Biron and A. Malhorta, eds., *XML Schema Part 2: Datatypes*, World Wide Web Consortium, 2 May 2001. [Online]. Available: http://www.w3.org/TR/xmlschema-2/